USENIX

DISTRIBUTED & MULTIPROCESSOR SYSTEMS (SEDMS III)

SPRING 1992

USENIX

SYMPOSIUM PROCEEDINGS

Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)

March 26 -27, 1992

Newport Beach, California

ISBN 1-880446-41-3

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.

SEDMS III

Symposium on Experiences with
Distributed and Multiprocessor Systems

Sponsored by
The USENIX Association
and
The Software Engineering Research Center (SERC)

In cooperation with:
ACM Special Interest Group on Architecture (SIGARCH)
ACM Special Interest Group on Data Communication (SIGCOMM)
ACM Special Interest Group on Operating Systems (SIGOPS)
ACM Special Interest Group on Software Engineering (SIGSOFT)
IEEE-CS Technical Committee on Design Automation (TCDA)
IEEE-CS Technical Committee on Distributed Processing (TCDP)
IEEE-CS Technical Committee on Operating Systems (TCOS)
IEEE-CS Technical Committee on Software Engineering (TCSE)

March 26-27, 1992
Neewport Beach, California

Introduction

This is the third Symposium on Experiences with Multiprocessor and Distributed Systems. The first was originally intended to be a workshop, held in October 1989. The level of response and quality of submissions resulted in a very high-level workshop that we consider to have been the first symposium. A second SEDMS was help in Atlanta, in March of 1991.

Once again, the theme of this Symposium is *experiences*. We have great respect for the role of theory and design in the development of computer systems, but we also recognize the unique insight that only real experience can provide. This Symposium continues to be a forum for reporting lessons learned during the development of real multiprocessor systems. The papers in this proceedings recount extensive experimentation, significant amounts of trial and error, careful thought, and some amount of serendipity. The program committee sifted through a number of very interesting papers, accepting only two out of every five submitted, to develop this program. We hope that the information shared in these works will help others in their continuing efforts.

This is the last SEDMS that we (George and Spaf) will be chairing. SEDMS IV is already being planned for spring of 1993. Peter Reiher of the Jet Propulsion Lab will be the general chair of SEDMS IV, and David Cohn of the University of Notre Dame will be the program chair. We encourage you to give them your suggestions for SEDMS IV, and to plan on submitting something for their program.

These three conferences have been both a great deal of work and a lot of fun. The continuing support of the Usenix Association has made the whole task possible. Continuing support by the Software Engineering Research Center has made the work of the program chair easier and helped us expand the audience for this event. We also appreciate the continuing association with the ACM and the Computer Society of the IEEE.

Our thanks to the hardworking members of our program committee (listed on the next page), and to all the reviewers who aided them in their reading and decision-making: H. Brams, L. Gossuin, F. Gridelet, C. Jacqmot, Thomas Joseph, James Kempf, Alan Langerman, Brian Lewis, M. Lobelle, Peter Madany, David V. Pitts, Bill Schilit, Carl Tait, Ping Sheng Tseng.

We greatly appreciate the hard work, advice, and efforts on our behalf by Ellie Young, Carolyn Carr, and Judy DesHarnais of Usenix. Georgia Conarroe, Shirley Shrum, and Daloris Williamson of Purdue proved invaluable (again) in helping keep Spaf on track with the program committee tasks and helping with multiple mailings. And thanks especially to all the people who took the time and effort to contribute papers to the symposium, and to come to Newport Beach in March to be with us. Thank you — we hope you enjoy it!

George Leach
General Chair

Gene Spafford
Program Chair

The Program Committee

Program and Table of Contents

Symposium on Experiences with
Distributed & Multiprocessor Systems (SEDMS) III
March 26 - 27, 1992

System Support for High Performance Multiprocessing [†]

Edward D. Lazowska
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195
lazowska@cs.washington.edu

Abstract

This paper surveys recent and ongoing work at the University of Washington concerned with system support for high performance multiprocessing. Three principal areas are addressed.

The first, and by far the most detailed, is the appropriate division of labor between the kernel and the user-level on shared memory multiprocessors. Our thesis here is that significant improvements in performance and flexibility can be achieved by moving functionality out of the kernel.

The second is support for sharing on wide-address machines. Our thesis here is that addressability and protection are orthogonal, and that with enough addressing bits and the right operating system design, it is possible to have an efficient and safe single-address space system that encourages flexible sharing.

The third is parallel/distributed computing. The goal is to build scalable high performance systems out of "commodity parts". The obstacles include devising a suitable programming model, and achieving low latency (as well as high bandwidth) communication.

1. Introduction

This paper surveys recent and ongoing work at the University of Washington concerned with system support for high performance multiprocessing. Three principal areas are addressed: system support for shared memory multiprocessors, for wide-address systems, and for parallel/distributed computing.

The common theme in this work is that newer architectures, and particularly multiprocessor architectures, can benefit from re-thinking the division of labor between the kernel and the user-level. On a uniprocessor, the kernel is responsible for address spaces, communication, resource allocation, threads, and thread scheduling. On a shared memory multiprocessor, the kernel should be left in charge of address spaces and resource allocation, but communication, threads, and thread scheduling should be implemented at user level through runtime support in each address space. Furthermore, each address space should have direct control over certain key aspects of virtual memory management and file management.

[†] The work discussed in this paper was done with Hank Levy, John Zahorjan, and our students, particularly Tom Anderson, Miche Baker-Harvey, Brian Bershad, Jeff Chase, and Chandu Thekkath. It was supported by the National Science Foundation, Digital Equipment Corporation, the Washington Technology Center, and Apple Computer.

It is important to understand that while "micro-kernel" designs such as Mach 3.0 take the important step of moving the file system, the virtual memory system, etc., out of the kernel to user-level servers, they leave cross-address space communication, threads, and thread scheduling in the kernel. The structural differences between monolithic kernels, micro-kernel designs, and the sort of "nano-kernel" that we advocate is illustrated in Figure 1 below.

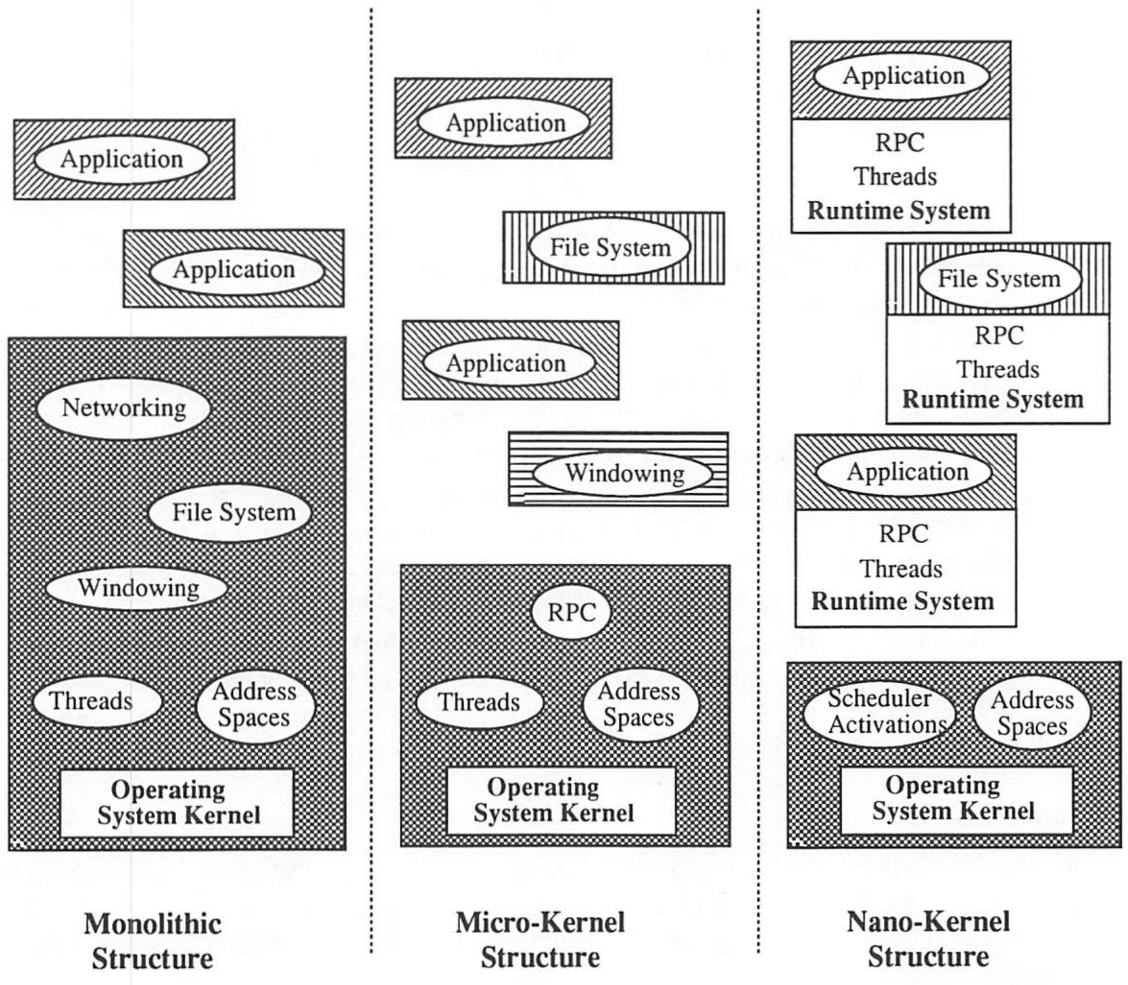


Figure 1: Monolithic, Micro-Kernel, and Nano-Kernel Structures

System Support for Shared Memory Multiprocessors

Current multiprocessor operating systems such as Mach [Tevanian et al. 1987], Topaz [Thacker et al. 1988], Chorus [Rozier et al. 1988], and V [Cheriton 1988], while adding primitives for parallelism, tend to retain the conventional structures that have been developed over decades of concentration on optimizing sequential computer performance. We believe that effective parallel computing can benefit from fundamentally different operating system structures and algorithms in several essential areas.

We have explored new approaches to locking [Anderson 1990], thread management [Anderson, Lazowska & Levy 1989], and cross-address space communication [Bershad et al. 1990, 1991]. We have prototyped a new kernel interface that makes it possible to combine the performance of user-level threads with the functionality of kernel threads [Anderson et al. 1992]. We have examined scheduling policies that exploit these mechanisms [Zahorjan & McCann

1990; McCann, Vaswani & Zahorjan 1990; Vaswani & Zahorjan 1991]. Finally, we have conducted a preliminary exploration of user-level virtual memory management [McNamee & Armstrong 1990].

The conclusion of this work is that while on a uniprocessor, performance is usually improved by placing functionality in the kernel, on a multiprocessor the opposite often is the case. The easiest area in which to argue this point is that of threads. Kernel threads offer an order-of-magnitude performance advantage over kernel processes, but user-level threads offer another order-of-magnitude advantage over kernel threads. (See Table 1.) There are two reasons for the performance advantage of user-level threads. First, if threads are supported in the kernel, then kernel calls are required to create, destroy, synchronize, and schedule them; kernel calls are expensive. Second, if threads are supported in the kernel, then they must be fully general (e.g., scheduled using a preemptive priority discipline) and fully safe. Beyond performance, an additional advantage of supporting parallelism at the user level is that different models of parallelism can be used in different address spaces: threads here, compiler-scheduled parallelism there [Polychronopoulos & Kuck 1987], futures one place [Halstead 1985], Chores another [Eager & Zahorjan 1991].

Operation	FastThreads user-level threads	Topaz kernel threads	Ultrix kernel processes
Null Fork	34	948	11300
Signal-Wait	37	441	1840

Table 1: Thread Operation Latencies (μsec. on a CVAX)

Cross-address space communication is another area where the advantages of a user-level implementation are clear. When a client in one address space invokes a server in another, both data and control must be transferred. In a traditional system architecture, each is transferred through the kernel. Even on a uniprocessor, performance can be improved by passing data at user level through carefully mapped shared memory [Bershad et al. 1990]. However, since there is only one processor and only the kernel can move this processor from one address space to another, the kernel must be involved in the transfer of control. On a multiprocessor, the presumption is that the server address space has processors available to execute the call, and that the client address space has additional threads to which the calling processor can be re-assigned cheaply. Both data and control can be transferred at user level, improving performance and providing compatibility with user-level threads [Bershad et al. 1991].

Unfortunately, the user-level management of parallelism suffers functionality and performance shortcomings in the face of "real world" events such as page faults, I/O, and processor preemption. This, we have demonstrated, is not inherent in the user-level management of parallelism, but rather is a consequence of the fact that kernel threads are not quite the right abstraction on top of which to build. Our new kernel interface, *scheduler activations* [Anderson et al. 1992] (similar work was done by [Marsh et al. 1991]; some of the same problems were attacked differently by [Edler et al. 1988] and [Black 1990]), has three key performance attributes: when an application makes minimal use of kernel services, it runs as quickly as with traditional user-level threads and much faster than with kernel threads; when an application requires kernel involvement because it does I/O, it performs better than either traditional user-level threads or kernel threads; in the presence of multiprogramming-induced kernel events such as processor preemption, it performs better than either traditional user-level threads or kernel threads. (See Table 2 for the speedup of two copies of a parallel solution to the N-body problem [Barnes & Hut 1986] being multiprogrammed on a 6-processor machine.)

Topaz kernel threads	FastThreads user-level threads	Scheduler Activations
1.29	1.26	2.45

Table 2: Speedup for Each of 2 Copies of the N-Body Problem Multiprogrammed on a 6-Processor System

We should note that scheduler activations appear to be suitable, although perhaps not necessary, on uniprocessors, and that user-level threads offer clear advantages in the uniprocessor domain. So the correct conclusion would appear to be that multiprocessors cry out for new structuring techniques, which if designed well may turn out to be useful or at least usable on uniprocessors as well.

We also should note that user-level control, which tends to reduce the number of system calls and context switches and improve locality, becomes increasingly advantageous as processor performance increases. It is well-documented that application code performance has benefited much more from modern RISC architectures than has operating system code performance [Ousterhout 1990]. For example, relative to a CVAX, a null system call runs 1.3 times as fast on an 88000, 1.8 times as fast on an R2000, and 1.0 times as fast on a SPARCstation 1+; the SPECmark advantages of these systems, though, are 3.5, 4.2, and 4.3 respectively [Anderson et al. 1991].

The "nano-kernel" system structure leads to a natural division of labor in control policies: processor allocation (the assignment of processors to address spaces) is the province of the kernel, while thread scheduling (the assignment of an address space's computational tasks to its processors) is the province of the runtime system in each address space. We have looked at two processor allocation problems for shared memory parallel machines: the appropriate processor allocation policy in a multiprogrammed environment, and the effect of "cache affinity" on processor scheduling.

Our group has studied several processor allocation policies [Zahorjan & McCann 1990; McCann, Vaswani & Zahorjan 1990]. The basic question here is whether the overhead of context switching is offset by the gains in processor efficiencies that result from moving a processor from an application that cannot currently use it to one that can. We provided evidence that overall performance is maximized by moving a processor as soon as it has no useful work to do.

In the domain of affinity scheduling, we concluded [Vaswani & Zahorjan 1991] that under the assumptions about system structuring outlined above, in which the role of the kernel in scheduling is simply processor allocation, the major kernel-related affinity consideration is in the selection of which kernel thread to restart when allocating a processor to an application. While our results indicate that affinity scheduling has little effect when employed in the kernel, this is not a general indictment of the idea. The kernel is a particularly difficult level at which to exploit affinity scheduling, largely because the opportunity to employ it there (i.e., at processor reallocations) occurs so infrequently.

System Support for Wide-Address Machines

The objective of this work is to reconsider operating system structure in light of the emergence of microprocessors with 64-bit addressing, an architectural trend that can already be seen in the MIPS R4000, the HP PA-RISC (the basis of the "Snake" workstations), the IBM RS/6000, the DEC Alpha, and other rumored processors.

We believe that the move to 64-bit addressing is far more important than a simple response to the continued need for extended addressability, such as occurred in the move to 32-bit architectures in the 1970s. Given a 64-bit address space, virtual addresses need *never* be reused; for example, if we allocate virtual address space at the rate of 1 megabyte every 10 milliseconds, a 64-bit address space will last 3,000 years! This can drastically change the way we think about and use address space. The move to 64-bit addressing allows us to reconsider issues such as the mapping of data onto long-term storage, the means of protection and cooperation, and the integration of distributed computing into both operating systems and programming languages.

We have noted that contemporary operating systems such as Mach, Topaz, Chorus, and V are in some respects caught between the old and the new. On one hand, they embody significant advances over older systems such as UNIX: they support newer requirements such as multiprocessing, and they provide better structure through improved inter-process communication, mapped files, and a micro-kernel design. On the other hand, they do not go far enough in moving functionality to the user level, and, more germane to this section, they are based on traditional models of "process" and "address space": direct sharing of memory is possible but not necessarily encouraged; it is difficult to use files directly to store structured data containing pointers without translating data on its way to or from long-term store; mapped files simplify the semantics of file access but don't change the type of information that can be stored in files; fundamentally, the use of memory, while made more flexible by user-level memory servers, still adheres to the traditional model of programs in independent address spaces communicating through messages or remote procedure call.

The major objective of our work in this area is to greatly enhance *sharing* and *cooperation* between complex applications executing on next-generation architectures. This research is being carried out in the context of a new operating system design, called Opal. The most important feature of Opal is its support for a *single global virtual address space* that maps all primary and secondary storage resources in a local area network. Conceptually, once a segment of the virtual address space is handed out, that segment is never reused, although the physical storage resources that back the segment are reclaimed. (Of course, for practical reasons there may be ways to reclaim virtual address space as well.) The crucial aspect of this addressing model is that the *meaning* or *interpretation* of a virtual address is independent of the entity issuing that address. Therefore, not only can programs share memory, but they can easily share complex data structures that include pointers, because the meaning of a pointer is always the same. In contrast, it is difficult to share such structures in conventional systems unless the cooperating parties agree in advance to map the structures into the same parts of their address spaces, which badly limits flexibility.

While sharing is greatly simplified by the single address space, we emphasize that this model does not entail a loss of protection. Previous single-address space systems, such as Pilot [Redell et al. 1980], relied exclusively on language-based protection. In Opal, the concepts of addressing and protection are independent at the kernel level; although any program can generate any virtual address, the translation of that address and access to the data are controlled. A program must first be given access to a segment of the address space. Management of the physical storage backing a virtual segment (called a "memory object" in the Mach terminology) is handled by memory servers, and access to servers is given through port capabilities. These capabilities are given out directly by the servers or exchanged between programs. Capabilities in Opal are similar to those in Amoeba [Mullender & Tanenbaum 1986] and Chorus [Rozier et al. 1988] in that they do not require kernel management (as, e.g., in Mach).

A system with the general structure described above would be related closely to capability-based or object-based systems that were prototyped in the past [Levy 1984]. Those systems were limited in scope, and in spite of hardware designed to help support capability-based protection, were too slow for general purpose use. However, it is our strong belief that (1) many of the objectives of these systems were to a large extent good, (2) their performance suffered for various technical reasons, and (3) we can now obtain most of the benefits at much lower cost.

System Support for Parallel/Distributed Computing

In the near future, high performance parallel computing will require a consideration of both parallelism and distribution; that is, truly scalable parallel systems will be built in a distributed way, with components connected via high-speed links. The individual components (i.e., the computational nodes) will be either uniprocessors or small-scale shared memory multiprocessors. There are many reasons why this architecture is likely. First, with current levels of integration, it is straightforward to connect several processors on a single board with a shared coherent memory. However, scaling this to even 20 processors, much less hundreds or thousands, is extremely difficult due to the bus bandwidth required. Second, the bandwidth of point-to-point links for even standard networks is approaching the gigabit per second range, and providing multiple links per node in a cube or mesh topology gives huge aggregate bandwidth, since multiple communications can occur simultaneously [Schroeder et al. 1990; Rodeheffer & Schroeder 1991]. Third, software environments and operating system support have advanced to the point where we can provide the illusion of a shared memory on a physically distributed hardware base. Thus, the decision of whether shared memory should be supported by hardware, by software, or by both is simply a cost/performance/complexity tradeoff. For scalable high performance systems, it is likely that at least some of the support for coherence will be provided by software.

Given this perspective, one key issue to explore involves programming environments that simplify the programming of distributed/parallel systems. A second key issue – one of enabling technology – is how to achieve low internode latency so that fine-grained parallelism can be utilized.

Our Amber system [Chase et al. 1989] is a small-scale prototype for the high performance systems we envision. Amber runs on a hardware environment consisting of shared memory multiprocessor nodes connected by an Ethernet local area network. Each node is a DEC SRC Firely multiprocessor [Thacker et al. 1988] consisting of 4 CVAX processors (plus one MicroVAX-II processor) and a coherent shared memory. Amber's objective is to use this distributed system as if it were an integrated multiprocessor; the research questions are how to make this easy for the programmer, and how to achieve good performance.

From an architectural viewpoint, the key issue is how to maintain *coherency* in the face of distribution. There is a spectrum of solutions. On one end is hardware-enforced coherency, for example, as provided by the Dash system [Lenoski et al. 1990]. At the other end is a system like Amber, in which coherence is done purely in software, in this case, at the object level. In the middle are schemes that use both hardware and software, for example, the Ivy shared virtual memory system [Li & Hudak 1989], in which virtual memory hardware is used to detect accesses to pages that are shared (replicated) on the network. There are many possible approaches, and careful evaluation of their tradeoffs is required, particularly in the face of new technology.

In the original Amber system, the distributed characteristics are fully visible to the programmer. Amber provides a global object space that is in some ways location independent and in other ways location dependent. Communication is location-invisible; if the target object happens to be local, the communication is fast, and if not, the communication, which occurs through an RPC-like mechanism, is slower. An object can move at any time under programmer control, and its address does not change because Amber implements a single address space across all of the nodes. More recently we have tried several strategies to simplify the programmer's task without sacrificing performance. The most recent relates directly to shared virtual memory as implemented in the Ivy system. Here, objects can be replicated on different nodes and can be accessed concurrently on those nodes, as long as the accesses only read object state. Once an operation to modify object state is requested, an invalidate-based coherency protocol is initiated by the system, which invalidates all but one copy. Following the completion of that operation, read-only operations on other nodes will cause more replicas to be created.

It is interesting to compare this latest version of Amber with Ivy. In Ivy, pages are the units of sharing and coherency; however, pages are invisible to programmers. The result is that false sharing can cause pages to thrash between nodes when accesses to different variables on the same page occur, even if those accesses do not require interlocks. On Amber, the units of sharing and coherency are objects, which are programmer-defined units; therefore, the programmer has control over the granularity of the coherency units. Another system related to Amber is Munin [Carter, Bennett & Zwaenepoel 1991], developed at Rice University. In the latest version of Munin, different coherency protocols are implemented for different objects, depending on the object's type. Thus, coherency can be tailored for different styles of sharing. Also, Munin is the first object-oriented distributed system to implement a weak coherency model. From Amber and its cousins, we can see some of the options in maintaining coherency in the face of distribution; the objective is to maintain coherency in the most cost-effective manner.

We noted that achieving low internode latency was another key challenge. While a high throughput interconnect allows rapid exchange of large blocks of data, it is the latency for small messages that will dictate the allowable granularity of a parallel computation. For example, if the latency were zero, then a fine-grained subdivision into threads could be made across node boundaries with frequent exchange of information between threads (and between nodes). If latency is high, then communication frequency must be reduced, and the computations into which an application is divided must be very course-grained. In this case, a problem would most likely be broken into only a small number of relatively heavy-weight computations. Depending upon the application, a loss of potential parallelism might result. That is, the more fine-grained parallelism the system permits, the more potential parallelism can be effectively utilized by the application; otherwise, the cost of creating, synchronizing, and communicating between fine-grained parallel components might greatly overshadow the advantage of that parallelism, and performance could easily suffer.

The current generation of networks is surprisingly bad in this regard. We implemented a stand-alone low latency RPC system on high performance workstations connected via FDDI, a fast current-generation network supporting 100 megabit per second transfers [Thekkath & Levy 91]. We measured the components of small RPC transfers, and tuned all of the possible software components. Our study shows that with modern software structures, software is unlikely to be the major barrier to achieving low-latency RPC communication. Instead, the network interface, and the controller in particular, constitutes the major problem. In fact, on our DECstation 5000s, small packet latency is *greater* for FDDI than for Ethernet, which provides 1/10th the bandwidth.

To minimize latency will require the design of software, hardware, and more importantly, the hardware/software interface for modern interconnects, that explicitly attacks the latency bottleneck. One area involves operating system structure. One would like to greatly reduce (or eliminate if possible) operating system intervention in application-to-application internode messages, without sacrificing security. This will require a new style of communications system interface. Second, new techniques, utilizing compiler support, can reduce the runtime system overhead inherent in current systems. Success in reducing cross-address space RPC overhead on single systems [Bershad et al. 1990] should extend into distributed environments.

Bibliography

[Anderson 1990]
T. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Distributed and Parallel Systems 1*,1, January 1990.

[Anderson & Lazowska 1990]
T. Anderson and E. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1990.

[Anderson, Lazowska & Levy 1989]
T. Anderson, E. Lazowska, and H. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers 38*,12, December 1989.

[Anderson et al. 1991]
T. Anderson, H. Levy, B. Bershad, and E. Lazowska. The Interaction of Architecture and Operating System Design. *Proc. 4th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[Anderson et al. 1992]
T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. To appear, *ACM Transactions on Computer Systems 10*,1, February 1992.

[Arnould et al. 1989]
E. Arnould, F. Bitz, E. Cooper, H.T. Kung, R. Sansom, and P. Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. *Proc. 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[Barnes & Hut 1986]
J. Barnes and P. Hut. A Hierarchical O(NlogN) Force-Calculation Algorithm. *Nature 324*, 1986.

[Bershad, Lazowska & Levy 1988]
B. Bershad, E. Lazowska, and H. Levy. Presto: A System for Object-Oriented Parallel Programming. *Software – Practice and Experience 18*,8, August 1988.

[Bershad et al. 1990]
B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems 8*,1, February 1990.

[Bershad et al. 1991]
B. Bershad, T. Anderson, E. Lazowska, and H. Levy. User-Level Remote Procedure Call. *ACM Transactions on Computer Systems 9*,2, May 1991.

[Birrell & Nelson 1984]
A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems 2*,1, February 1984.

[Black 1990]
D. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer Magazine 23*,5, May 1990.

[Chase et al. 1989]
J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. *Proc. 12th ACM Symp. on Operating Systems Principles*, December 1989.

[Cheriton 1988]
D. Cheriton. The V Distributed System. *Communications of the ACM 31*,3, March 1988.

[Daley & Dennis 1968]
R. Daley and J. Dennis. Virtual Memory, Processes, and Sharing in Multics. *Comm. of the ACM 11*,5, May 1968.

[Draves et al. 1991]
R. Draves, B. Bershad, R. Rashid, and R. Dean. Using Continuations to Implement Thread Management and Communication in Operating Systems. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Eager & Zahorjan 1991]
D. Eager and J. Zahorjan. Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing. Technical Report 91-08-05, Department of Computer Science & Engineering, Univ. of Washington, August 1991. Submitted.

[Edler et al. 1988]
J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. *Proc. USENIX Workshop on UNIX and Supercomputers*, September 1988.

[Felten & McNamee 1991]
E. Felten and D. McNamee. Improving Performance of Message-Passing Applications by Multithreading. Department of Computer Science & Engineering, Univ. of Washington, November 1991. Submitted.

[J. Faust 1990]
J. Faust. *The Performance Tuning of an Object-Oriented Threads Package*. Master's Thesis, Department of Computer Science & Engineering, University of Washington, March 1990.

[K. Faust 1990]
K. Faust. *An Empirical Comparison of Object Mobility Mechanisms*. Master's Thesis, Department of Computer Science & Engineering, University of Washington, April 1990.

[Faust & Levy 1990]
J. Faust and H. Levy. The Performance of an Object-Oriented Threads Package. *Proc. 4th ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1990.

[Feeley et al. 1991]
M. Feeley, B. Bershad, J. Chase, and H. Levy. Dynamic Node Reconfiguration in a Parallel-Distributed Environment. *Proc. 3rd ACM SIGPLAN Conf. on Principles and Practice of Parallel Programming*, April 1991.

[Gupta, Tucker & Urushibara 1991]
A. Gupta, A. Tucker, and S. Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. *Proc. 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1991.

[Halstead 1985]
R. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems* 7,4, October 1985.

[Jones & Rashid 1986]
M. Jones and R. Rashid. Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, October 1986.

[Karlin et al. 1991]
A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical Studies of Competitive Spinning on A Shared-Memory Multiprocessor. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Kung et al. 1991]
H.T. Kung, P. Steenkiste, M. Gubitoso, and M. Khaira. Parallelizing a New Class of Large Applications over High-speed Networks. *Proc. 3rd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, April 1991.

[LaRowe, Ellis & Kaplan 1991]
R. LaRowe, Jr., C. Ellis, and L. Kaplan. The Robustness of NUMA Memory Management. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Lenoski et al. 1990]
D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proc. 17th Annual Int'l. Symp. on Computer Architecture*, May 1990.

[Levy 1984]
H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[Li & Hudak 1989]
K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems* 7,4, November 1989.

[Lo & Gligor 1987]
S.-P. Lo and V. Gligor. A Comparative Analysis of Multiprocessor Scheduling Algorithms. *Proc. 7th Int'l. Conf. on Distributed Computing Systems*, September 1987.

[Marsh et al. 1991]
B. Marsh, M. Scott, T. LeBlanc, and E. Markatos. First-Class User-Level Threads. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[McCann, Vaswani & Zahorjan 1990]
C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Strategy for Multiprogrammed, Shared Memory Multiprocessors. Technical Report 90-03-02, Department of Computer Science & Engineering, Univ. of Washington, March 1990. Submitted.

[McNamee & Armstrong 1990]
D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. *Proc. USENIX Mach Workshop*, October 1990.

[McNamee & Felten 1991]
D. McNamee and E. Felten. NewThreads: Programming Support for Distributed Memory Parallel Programming. Department of Computer Science & Engineering, University of Washington, November 1991. Submitted.

[Moeller-Nielsen & Staunstrup 1987]
P. Moeller-Nielsen and J. Staunstrup. Problem-Heap: A Paradigm for Multiprocessor Algorithms. *Parallel Computing 4*,1, February 1987.

[Mohr, Kranz & Halstead 1991]
E. Mohr, K. Kranz, and R. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. on Parallel and Distributed Systems 2*,3, July 1991.

[Mullender & Tanenbaum 1986]
S. Mullender and A. Tanenbaum. The Design of a Capability-Based Operating System. *The Computer Journal 29*,4, 1986.

[Ousterhout 1982]
J. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proc. 3rd Int'l. Conf. on Distributed Computing Systems*, October 1982.

[Ousterhout 1990]
J. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proc. Summer 1990 USENIX Conf.*, June 1990.

[Polychronopoulos & Kuck 1987]
C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers C-36*,12, December 1987.

[Polychronopoulos 1990]
C. Polychronopoulos. Auto-Scheduling: Control Flow and Data Flow Come Together. Technical Report CSRD-TR-1058, Center for Supercomputing Research and Development, Univ. of Illinois, November 1990.

[Raj et al. 1990]
R. Raj, E. Tempero, H. Levy, N. Hutchinson, A. Black, and E. Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience 21*,1, January 1991.

[Redell et al. 1980]
D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An Operating System for a Personal Computer. *Comm. ACM 23*,2, February 1980.

[Rodeheffer & Schroeder 1991]
T. Rodeheffer and M. Schroeder Automatic Reconfiguration in Autonet. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Rozier et al. 1988]
M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Leonard, S. Langlois, and W. Neuhauser. Chorus Distributed Operating Systems. *Computing Systems 1*,4, 1988.

[Schroeder et al. 1990]
M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links. Technical Report 59, DEC Systems Research Center, April 1990.

[Scott, LeBlanc & Marsh 1990]
M. Scott, T. LeBlanc, and B. Marsh. Multi-Model Parallel Programming in Psyche. *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, March 1990.

[Squillante & Lazowska 1991]
Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. To appear, *IEEE Trans. on Parallel and Distributed Systems.*

[Tevanian et al. 1987]
A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Threads and the Unix Kernel: The Battle for Control. *Proc. 1987 USENIX Summer Conf.*, 1987.

[Thacker et al. 1988]
C. Thacker, L. Stewart, and E. Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers 37*,8, August 1988.

[Thekkath et al. 1991]
C. Thekkath, D. Eager, E. Lazowska, and H. Levy. A Performance Analysis of Network I/O in Shared-Memory Multiprocessors. Department of Computer Science & Engineering, Univ. of Washington, October 1991. Submitted.

[Thekkath & Levy 1991]
C. Thekkath and H. Levy. Limits to Low-Latency RPC. Department of Computer Science & Engineering, Univ. of Washington, July 1991. Submitted.

[Tucker & Gupta 1989]
A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors. *Proc. 12th ACM Symp. on Operating Systems Principles*, December 1989.

[Vandevoorde & Roberts 1988]
M. Vandevoorde and E. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *Int'l. Journal of Parallel Programming 17*,4, August 1988.

[Vaswani & Zahorjan 1991]
R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. *Proc. 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Weiser et al. 1989]
M. Weiser, A. Demers, and C. Hauser. The Portable Common Runtime Approach to Interoperability. *Proc. 12th ACM Symp. on Operating Systems Principles*, December 1989.

[Wilson 1991]
P. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *ACM SIGARCH Computer Architecture News 19*,4, June 1991.

[Wulf et al. 1974]
W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Comm. ACM 17*,6, June 1974.

[Zahorjan & McCann 1990]
J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. *Proc. 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1990.

[Zahorjan, Lazowska & Eager 1988]
J. Zahorjan, E. Lazowska, and D. Eager. Spinning Versus Blocking in Parallel Systems with Uncertainty. *Proc. Int'l. Symp. on Performance of Distributed and Parallel Systems*, December 1988.

[Zahorjan, Lazowska & Eager 1991]
J. Zahorjan, E. Lazowska, and D. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 2*,2, April 1991.

Lessons from Implementing Active Objects on a Parallel Machine

Riccardo Capobianchi Rachid Guerraoui
Agnes Lanusse Pierre Roux

CE Saclay DEIN/SIR
91191 Gif sur Yvette Cedex France
E_mail : capobianchi@leti.cea.fr guerraoui@leti.cea.fr
lanusse@leti.cea.fr

Phone : (+33)(+1) 69.08.82.94
Fax : (+33)(+1) 69.08.83.95

Abstract

We present a practical experience concerning the implementation of a concurrent object-oriented language, called AKC++ (Actor Kernel in C++), on a parallel machine. Object-oriented languages characteristics are proving to be very useful in several areas where more conventional approaches have encountered many problems. Properties like encapsulation and inheritance make it easy to model any real-world entity. As those entities may exist and act concurrently, object-oriented languages show a great potential for use in designing and building parallel systems. Looking at objects as at independent active entities communicating through asynchronous message passing seems very suitable for writing parallel programs Our case study concerns the transfer of the language kernel we designed from a sequential environment, in which the parallelism was simulated, to a shared memory parallel machine. The porting of our kernel to a multiprocessor configuration permitted us to address different problems, mainly due to an unmanageable proliferation of processes and to a waste of allocated computational resources. We propose a pragmatic solution based on dynamic task allocation and explicit context switching mechanisms. Our approach seems well suited for many object-based concurrent languages because it does not depend on a specific active object model.

1. Introduction

The integration of parallel programming with object-oriented languages appears to be a natural way to exploit the power of new technologies. These advances have made it possible to build up parallel machines, as well as networks of computers where the executions may be carried out in parallel. On the other hand, object-oriented languages have shown to be a major step towards such desirable features as data abstraction, software modularity, code reusability and readability [Mey 88].

There are several ways to combine object-oriented languages with parallelism [Pap 89]. The most obvious way is to add the traditional concept of process to the language. If we look at the essence of the object paradigm, we see that a system to be modelled is subdivided into objects, where each object is an integrated unit of data and procedures acting on these data. This uniform view may lead to a tight integration of objects and processes: the message passing model of communication between autonomous objects provides a good abstraction for parallel program execution. An object may thus be viewed as an active autonomous module (active object) containing its own processing resources. Agha actors [Ag 86] are active objects which execute concurrently and communicate through asynchronous message passing. Starting from the Agha model of computation, we have designed and implemented AKC++ : an Actor Kernel in C++ [Lan 91]. Perhaps the ideal architecture for such an active object system would be one in which the number of processors is comparable to the number of active objects and where a single processor would be flexible enough to run an object [El 89]. Even though future parallel machines will probably support such architecture, now they are not.

In this paper we present a case study concerning the transfer of AKC++ from a sequential environment, in which the parallelism was simulated, to a shared memory parallel machine. This transfer pointed out many practical problems such as the proliferation of thousands of either statically or dynamically (at run-time) created active objects. We claim that those problems as well as the solutions we propose are not specific to AKC++ but may be taken into account in many cases which involve moving from a prototyped simulated active object system to a parallel environment supporting real applications.

The paper is organized as follows: Section 2 introduces concurrent object-oriented programming and the actor model of computation, as well as the AKC++ language kernel we designed. Section 3 presents a common way of simulating parallelism, and discusses the problems we encountered when porting our language kernel from a parallelism simulation to a real concurrent situation. Those drawbacks may be critical in many concurrent object-oriented languages implementations. In section 4 we propose some pragmatic solutions. Section 5 concludes with some final remarks and the illustration of our future research directions.

2. Designing a concurrent object-oriented language

2.1 From objects to active objects

There are at least two reasons to consider features of concurrency in a programming language. First, the real world is concurrent and many information systems are intended to model real world entities. Second, such features make it easier to exploit the power offered by parallel architectures.

The essence of object-based programming is encapsulation of local data. Each object encapsulates some local state that may be accessed only by the methods that are somehow associated to the object (usually via a class definition). Objects may access the local state of another object only by requesting the recipient of a message to execute some method. The message passing metaphor treats objects as autonomous active entities that synchronize and exchange information with one another only by explicitly sending messages. This model combines access to data and synchronization: there is no distinction between passive data and active processes but only one single construct named active object.

Active object models [Weg 87] may be classified into sequential, quasi concurrent and concurrent according to their granularity of parallelism:

- Sequential : they possess one thread of control through which all the received messages are treated one at a time. In an asynchronous message passing environment, objects may own some kind of queue which records the arriving messages. Objects are free to execute the messages in a random order as long as these are executed one at a time. Internal consistency of objects is implicitly protected against concurrent modifications. Ada [ANSI 83] tasks and POOL-T [Am 87] objects are examples of sequential objects.
- Quasi-Concurrent : as in the previous model, one thread per object is active at a time. However, an object may interrupt its current active thread and turn its attention to another thread. This is very useful when an object sends a message and performs another work while waiting for a reply. Hybrid [Nier 87] domains, for example, may use delegation to interfere threads. In ABCL [Yon 90], an object that is treating an ordinary message can be interrupted to treat an express message. Such interfering must however be carefully done because an interrupted thread may leave an object in an inconsistent state.
- Concurrent : many threads of control may be active in an object at the same time. ARGUS [Lis 88] guardians and PO [Cor 87] objects associate threads with each incoming message. Such interference of concurrent threads may be controlled in different manners. Guardians internal concurrency control is guaranteed by atomic internal data types while PO provides both a priori and internal concurrency control. Also the actor model [Ag 86] belongs to this category.

Actors are concurrent processes communicating through asynchronous message passing. Each actor is identified by its address that may be communicated to other actors just as any other value. Such address communication provides a simple mechanism for dynamic reconfiguration [Kaf 90].

An actor has a behaviour script and a set of acquaintances. The behaviour script describes the way the actor reacts to messages while acquaintances represent reachable actors' addresses. Each time an actor accepts a message it may perform one of its primitives. The three basic actor primitives correspond to the creation of a new actor (*create-actor*), the invoice of a message to an actor (*send-message*), and the replacement of the current behaviour with a new one (*become*). The *become* operation allows actors to change state by determining the behaviour that will treat the next communication to process (a behaviour receives and treats exactly one message). An actor carries out its primitive actions concurrently [Ag 90].

2.2 AKC++ active objects

Starting from the actor model of computation [Ag 86], we have designed a kernel of a concurrent object-oriented language (AKC++, Actor Kernel in C++). This kernel uses active objects (*aobjects*) as computational agents that evolve and cooperate in a parallel way. Each *aobject*'s behaviour is built from a list of methods definitions while acquaintances are represented by state variables. An *aobject* is identified by the address of its mail queue, that is also used to buffer all the incoming messages. A message specifies a request of a method execution to the addressed *aobject*.

Confronted with the actor model, AKC++ has some specific properties:

- each active object may execute a sequential code before changing its behaviour: the methods' instructions are not executed in parallel like they were in the original model. Messages are serialized in the mail queue (*mail-box*) associated to the *aobject* (fig. 1). AKC++ *aobjects* are still concurrent because of the intra-object parallelism introduced by the *become* construct. When it reaches such a statement the *aobject* duplicates its state and, if its *mail-box* is not empty, executes another message. Thus the *become* operation unlocks the *aobject* status, because it allows to treat the next message (fig. 2).
- an *aobject* may receive the result of its request by creating a reply box (*wait-box*) and including the address of this reply box in the request message. The *wait-box* acts as a synchronization point when the client *aobject* tries to read a result which has not yet arrived. When the reply message arrives the *aobject* execution is resumed. A method execution may thus be blocked if it tries to get a result from an empty *wait-box*.

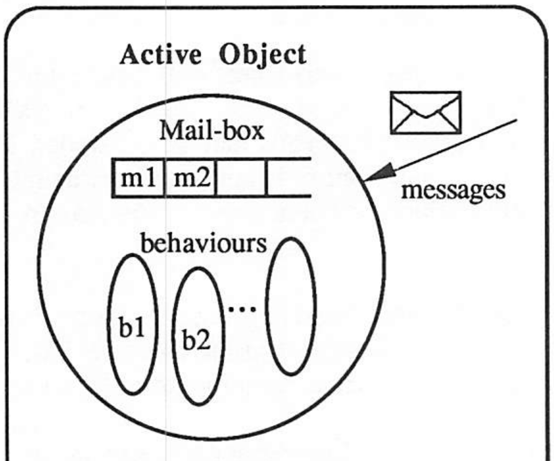


fig. 1 - Active Object Architecture

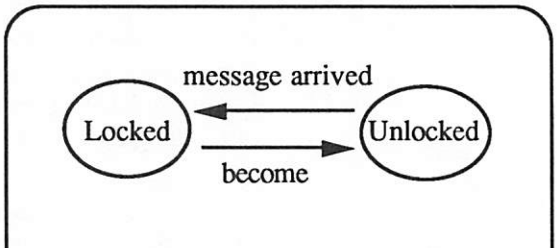


fig. 2 - Active Object Status

In AKC++, the parallelism is both inter-object, due to the fact that each *aobject* has its own processing resource, and intra-object [Cor 89], permitted by the use of the *become* operation that allows to split an *aobject*'s execution flow in two or more (the *aobject* whose executing method performed a *become* may continue its computations while a new behaviour is free to process the next incoming-message specified method).

3. Porting AKC++ on a parallel machine

3.1. The first prototype

The first implementation of the AKC++ language, written in C++ and running on a Sun workstation under Unix BSD 4.0, simulates the parallelism through the use of the Sun Lightweight Process Library [Sun 90].

A lightweight process (from now on *lwp*) represents a thread of control that is not bound to an address space. *Lwps* operate more efficiently than ordinary processes because they communicate through shared memory instead of a file-system or IPC (Inter Process Communication). The cost of creating tasks and managing communications is thus substantially reduced. All threads live inside a single Unix process; scheduling is by default priority based, non-preemptive within a priority. A set of primitives allows one to manipulate *lwps*, to control their scheduling and communication.

In our language each *aobject* has its own attached *lwp*: in fact, to each new *aobject* creation corresponds the creation of a new thread (*lwp*) that is used to execute the method associated to the first message the *aobject* will receive. If this method contains a *become* operation, a new *lwp* will be created and associated to the specified new behaviour to treat the next message; the current thread will die out at the end of the current method's computation. If no *become* operation has been performed, the same *lwp* and the same behaviour will be associated with the treatment of the next message request: this corresponds to performing an implicit *become-self* operation at the end of each method execution.

New *aobjects* with their associated *lwps* may be created during the initialization phase as well as during a method execution as a result of a *create-actor* operation. However, if an *aobject* has no message to treat in its mail queue, the associated thread will be blocked until a message arrives. A round-robin time-sliced scheduler has been implemented through the use of *lwp* primitives to simulate parallel execution of a set of threads with the same priority level (as is the case with *lwps* attached to *aobjects*).

3.2. Parallel implementation

3.2.1. The Dune machine

Our next step has been to transfer our language kernel on a real multiprocessor architecture, in our case a Dune 3000 computer running with the Dune-ix operating system. This machine has a shared memory, and can be equipped with a maximum of four processors, each using a local cache memory to access the shared memory.

The operating system is Unix V.3 compatible, with an augmented set of primitives. The task scheduler allocates the processors realizing a dynamic load balancing; this is done in a way that is completely transparent to the user, who can program without confronting to the multiprocessor aspects. The tasks to be executed are chosen from a single queue ordered by logic priority: if the machine is equipped with *n* processors, at each moment the *n* processes with the highest priorities are executed on the *n* processors [Bas 90].

3.2.2. Porting AKC++

The Dune machine does not support the lightweight processes library, so when porting our language we had to find out which kind of thread we could associate to each of our *aobjects*. To avoid using conventional Unix processes (with their burden of communication operations), we decided to use a sort of "lighter" processes issued from an *afork* operation. This system call creates a twin process to the caller, that shares the same data segment and heap having just a new stack of its own; the two processes can thus run concurrently and communicate via shared memory.

Being in a real parallel environment we could suppress our *lwp* scheduler and rely on the machine scheduler. We obtain an architecture in which each executing *aobject* (current behaviour / message specified method) has an attached process and processor. Each executing *aobject* runs concurrently with other *aobjects*, possibly with other copies of the same *aobject* bound to new *become*-specified behaviours and the corresponding new messages. Communications between *aobjects* are messages passed through shared memory; synchronization is controlled with the use of semaphores.

In the Sun parallel simulation case, all the created *lwps* evolved inside a single Unix process; at the end of the application execution, their existence could be terminated with a single command (at the *lwp* library level or at the shell level). On the parallel machine, each "light" process is equivalent to a conventional Unix process, and to terminate it we must execute an explicit (*kill*-style) shell command. In our language every *aobject*, once created, has always at least one attached process waiting for a message to treat; this means that at the end of the computation all the created *aobjects* will remain as pending processes in the system, since they have no way of knowing if they should terminate or if they might receive other messages.

3.2.3. The process proliferation problem

When we come to think about building real-sized applications with our language kernel, we become concerned with writing programs that may create, statically or dynamically, hundreds or thousands of *aobjects*. According to our model, every time one of these *aobjects* is created a new "light" (*afork*-issued) process is created and attached to it; its role is to wait for a message and to execute the associated method. For the most of their time, these processes have really nothing to do: they just remain still waiting for a message to treat. It is only when a message arrives that the process can become active and perform some operation.

During a method execution other processes may be created if the method contains either a *create-aobject* or a *become* operation. This leads to the creation of a very large number of processes, each of it having an associated stack and competing for a processor attribution.

The above considerations force us to confront with the physical limits of a real machine:

- the allocation of thousands of stacks for thousands of *aobjects* would quickly eat out all the available memory;
- an unlimited process proliferation would rapidly exceed the limits imposed by the machine (on the maximum number of created processes);
- even without considering the above mentioned limitations, the fact of having a fixed number of available processors (four in our case) with thousands of processes contending for their use (thus resulting in thousands minus four blocked processes) does not seem reasonable at all.

4. Pragmatic enhancements

4.1. Pending processes

We have modified our language implementation by removing the implicit *become-self* operation that was performed at the end of a method execution if no explicit *become* was performed. It now becomes the task of the programmer to decide whether an *aobject* should continue to live after the execution of a method, in which case it must include a *become-self* instruction in his method definition. This permits us to avoid the possibility that *aobjects* which are meant to treat only one message would continue to live indefinitely in the system, along with their associated pending process, after the end of the computational flow. We still however need an *aobject* garbage collector because the problem is not completely solved: unreachable *aobjects* may still be alive.

Removing the implicit *become-self* operation imposes the termination of the associated process at the end of the execution of the current method; a new process to treat the following incoming message will be created only if the method contains a *become* or *become-self* operation. This process will compete with the others for the attribution of a processor. In this way, each method execution is associated with exactly one process.

4.2. The processes pool

Taking into account the fact that we have at our disposal a limited set of processors, it seems natural to fix a limit for the number of processes that can be created, and to tune this limit to the number of available processors. Our solution for the process proliferation problem is to create, during the initialization phase, a fixed-number pool of *afork*-issued "light" processes that are dynamically attached to the *aobjects* which can perform some computation.

We introduce the notion of task, that is an entity representing an *aobject* with an executable method. The dynamic task allocation is achieved through a "ready queue" containing only the "ready" tasks. Each process of the pool executes a loop during which it examines the queue, gets the first task it finds and executes it (fig. 3). A task execution means the execution of a specific message-specified method on a specific *aobject.*

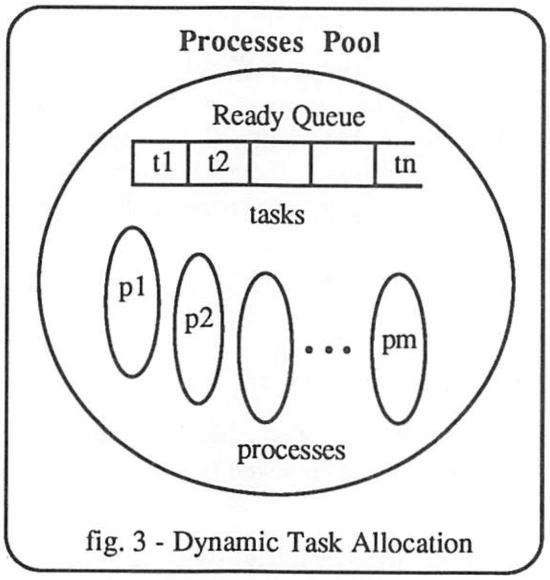


fig. 3 - Dynamic Task Allocation

An *aobject* is considered "ready" either if it performs a *become* operation and its *mail-box* contains a message to treat, or if it receives the first message after its creation. The "ready" *aobject* can thus be associated to a ready task to be put in the ready queue.

Three cases involve the creation of a task and its insertion in the ready queue:

- the *send-message* operation creates a ready task associated to the receiving *aobject* if this one was newly created (it is receiving its first message);
- the *send-message* operation creates a ready task associated to the receiving *aobject* if this one has already executed a *become* operation (and is not already in the queue); this may introduce intra-object parallelism if the method that performed the *become* is still executing and the task is actually taken from the ready queue by a process;
- the *become* operation creates a ready task associated to the *aobject* performing it if the *aobject*"s mail queue contains some message; this introduces intra-object parallelism when there is a free process that gets the task from the ready queue.

With this configuration it becomes easy to implement a resource collector for our processes: it is enough to have a running process that periodically examines the ready queue. When the queue is empty, and no processes of the pool are attached to a task, the computations are over because no more communications can be passed and no *aobject* can become active anymore. The resource collector can thus kill the processes in the pool and exit.

4.3 Explicit context switching

The proposed task allocation architecture still presents a problem: what happens if an executing task blocks itself, i.e. because the *aobject*'s method is waiting on a *wait-box* (it needs a reply from another *aobject*), keeping the attached process uselessly occupied? We can see that this produces unwieldy inefficiencies and may lead to a deadlock situation, in which all the processes in the pool are blocked waiting on events that could be produced only by ready tasks having no chance to obtain a processing resource. The original implementation avoided this kind of deadlocks because the number of processes was not a priori limited.

If we don't want to drop the waiting-for-a-reply blocking operation, introducing costly continuation mechanisms [Ag 86], we can imagine associating a switching context operation to each blocking operation. This means that when a method needs a result from one of its *wait-boxes* and that result isn't yet there, the context of the current executing method is saved; the associated process is detached from the task and gets back to the pool to execute the usual cycle.

We associate four components to each task:

1) **ao** : the *aobject*;

2) **m** : the method to execute;

3) **ctx** : the saved execution context; this component is not empty only if the task was previously running and was blocked because its method tried to read a result from an empty *wait-box*;

4) **st** : the task status; it may be "ready" (waiting to be allocated to a process of the pool), "running" (allocated to a process) or "blocked" (waiting for a result in a *wait-box*).

As was noted above, the ready queue contains only the tasks whose status is "ready". In addition to the three cases involving the insertion of a task in the ready queue (cited in subsection 4.2), a task may pass in the ready status if it was blocked (because its method tried to read an empty *wait-box*) and has just been unblocked (because some *aobject* put a result in that *wait-box*).

Every process of the pool executes the same infinite loop:

```
examine the ready queue;
if the queue is not empty
 then
 {
   get the first task out of the queue;
   if the task has a saved execution context
     then
     {
       put the task status to "running";
       restore the task execution context;
       empty the task execution context slot;
       continue the task execution;
     }
     else
     {
       put the task status to "running";
       start the task execution;
     }
 }
 else yield control to another process;
```

Two independent flags are used to manage the synchronization between tasks:

1) **PF "parallel-flag"** : this flag is bound to each *aobject*. If it is *off* it means that the task method has performed a *become* operation (or that the *aobject* has never executed a method), but there are no messages in the related *aobject* '*smail-box*. This flag represents the possibility of having intra-object parallelism (except for the first message treated by an *aobject*), because when it is *on* it means that a new task related to the same *aobject* has been inserted in the ready queue.

2) **SF "sync-flag"** : this flag is bound to each *wait-box*. If it is *off* it means that the task is blocked, because its method tried to read a result from that *wait-box* when this one was empty. This flag represents the synchronization mechanism between *aobjects* realized through the *wait-boxes*, because its value shows if a task is waiting for a result from another task. To each *wait-box* is also associated a reference to the related task.

During a task (and thus a method) execution, the operations that can affect the ready queue and the synchronization flags are the following:

- **create an *aobject*** : for the created *aobject*, *PF* is *off* ;

- **send a message *m* to an *aobject ao*** :
if the *aobject*'s *PF* is *off*
then insert in the ready queue a new task associated to *ao* and the method requested by *m*, with no saved execution context and status "ready"; put *PF* to *on* ;

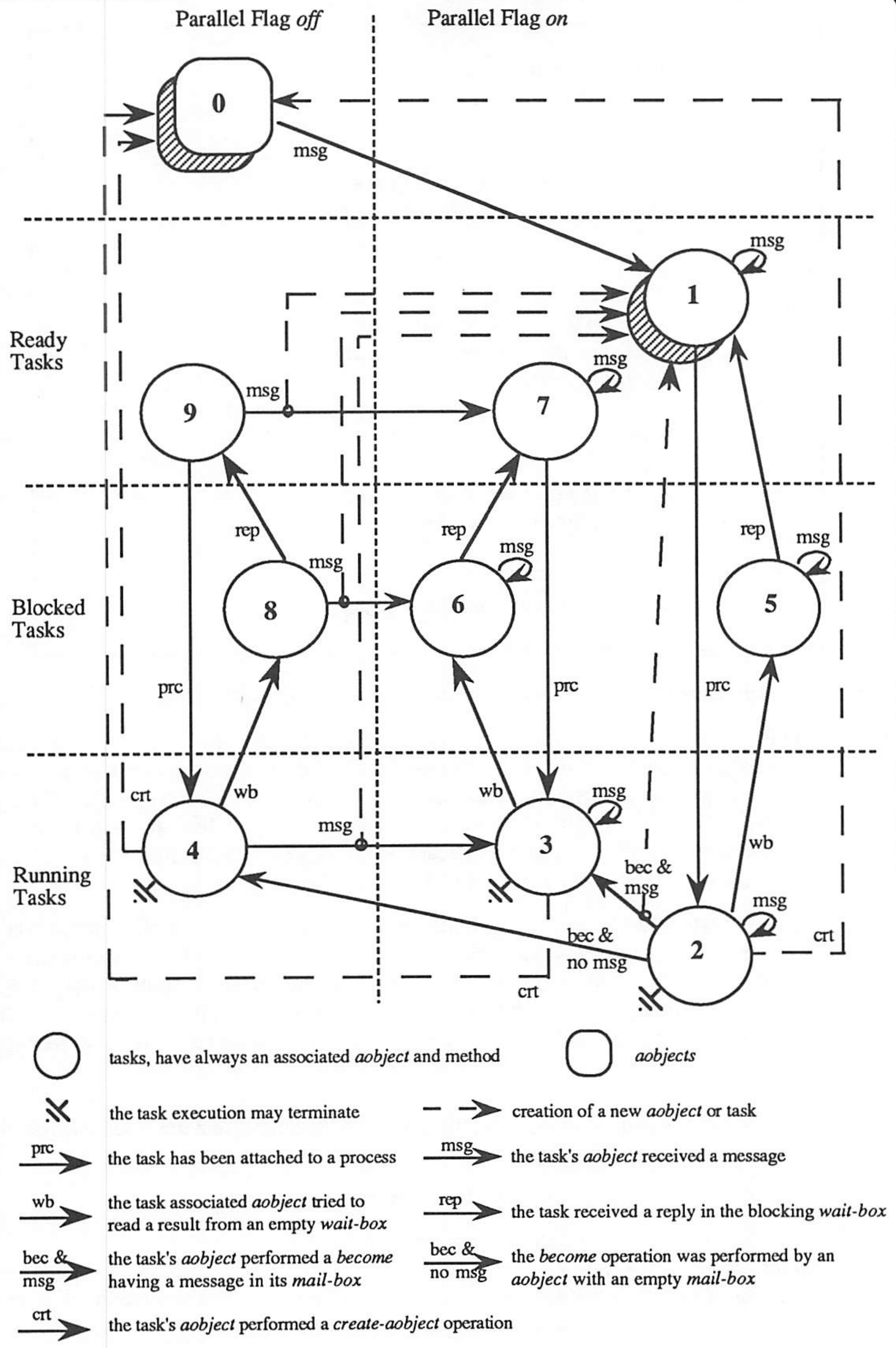

Parallel Flag *off*
Parallel Flag *on*
Ready Tasks
Blocked Tasks
Running Tasks
0
1
2
3
4
5
6
7
8
9
msg
rep
prc
wb
crt
bec & msg
bec & no msg
tasks, have always an associated *aobject* and method
aobjects
the task execution may terminate
creation of a new *aobject* or task
prc
the task has been attached to a process
msg
the task's *aobject* received a message
wb
the task associated *aobject* tried to read a result from an empty *wait-box*
rep
the task received a reply in the blocking *wait-box*
bec & msg
the task's *aobject* performed a *become* having a message in its *mail-box*
bec & no msg
the *become* operation was performed by an *aobject* with an empty *mail-box*
crt
the task's *aobject* performed a *create-aobject* operation

fig. 4 - Effects of method operations on tasks and *aobjects*

- ***become* an *aobject ao*** :
if the *mail-box* contains a message *m*
then insert in the ready queue a new task associated to *ao* and the method requested by *m*, with no saved execution context and status "ready";
else put *PF* to *off*;

- **read a result from a *wait-box*** :
if the *wait-box* is empty
then put the current task status to "blocked", put the *wait-box*'s *SF* to *off* , save the task execution context and return to the pool cycle ;

- **put a result in a *wait-box*** :
if the *wait-box*'s *SF* is *off*
then insert the related task (with its status put to "ready") in the ready queue, and put the *wait-box*'s *SF* to *on* .

Fig. 4 tries to show the relationships between a task status transitions, the operations which may be performed during methods' executions and the process allocations.

The above described configuration allows us to obtain an architecture which assures that only those *aobjects* which can really execute some computation are attached to a process, realizing an optimization for the allocation of computational resources.

We cannot still say that the number of processes we create in our pool should be equal to the number of available processors in our parallel machine, because we must take into account the fact that a process may execute a low-level blocking operation (such as an input/output system call), and that in this case the related processor would remain unoccupied having no other process to pick up. Thus a fair quota for the processes to insert into the pool seems to be around the double of the number of available processors.

5. Concluding remarks

The object paradigm enjoys properties such as uniformity, factorisation and dynamicity. It provides a good abstraction to many low level features that must be dealt with when building parallel applications. The integration of concurrency and object orientation is a very appealing conceptual view, allowing to introduce inter-object and intra-object parallelism.

In this paper we addressed some issues related to a practical experience concerning the porting of a concurrent object-oriented language kernel from a sequential environment, in which the parallelism was simulated, to a shared memory parallel machine supporting real applications. Those issues are mainly due to a proliferation of thousands of created processes and to a waste of allocated computational resources; we believe that those problems may be encountered in other similar cases and that they can severely impact on a language performances if they are not taken into account. We presented a pragmatic solution based on dynamic task allocation and explicit context switching mechanisms which may be applied to many others object-based languages. The solution that has been implemented within the AKC++ kernel is completely user transparent: the programmer has not to take care of the processors and processes allocation policy.

We are currently investigating the possibility of confining all the system-level blocking operations that can be performed during a method execution into some special *blocking aobjects*. This would allow us to write methods whose execution could be stopped only after explicit blocking commands, and thus to further optimize the proposed resource allocation mechanism. A synthesis on the implementation choices made in the sequential and parallel environments has been done and will lead to a comparative performance evaluation.

Acknowledgements

The authors are deeply indebted to J. Delcoigne for the many ideas that sprung during many discussions, as well as for its useful comments.
This work has been partially founded by the ESPRIT Project 2434.

References

[ANSI 83] American National Standards Institute, Inc. - The Programming Language ADA Reference Manual - LNCS 155 - Springer-Verlag - 1983.

[Ag 86] Agha G. - Actors: a Model of Concurrent Computation in Distributed Systems - The Mit Press - 1986.

[Ag 90] Agha G. - Concurrent Object-Oriented Programming - Communications of the ACM - Vol. 33, No. 9 - September 1990.

[Am 87] America P. - POOL-T: a Parallel Object-Oriented Language - in Object-Oriented Concurrent Programming - ed. A. Yonezawa and M. Tokoro - The MIT Press - 1987.

[Bas 90] Bastien M., Bras D., Chazal G. and Delcoigne J. - The Dune-ix Real Time Operating System - CEA / DEIN Report SIR 90/93 - 1991.

[Cor 87] Corradi A. and Leonardi L. - How to Embed Concurrency within an Object Environment: Parallel Objects - Proceedings of the International Conference on Parallel Processing and Applications 1987 - Elsevier Science Publishers B.V. - September 1987.

[Cor 89] Corradi A. and Leonardi L. - Concurrency: the Neglected Issue of Object Systems - Proceedings of the Third International Parallel Processing Symposium - 1989.

[El 89] Ellis C. and Gibbs S. - Active Objects: Realities and Possibilities - in Object-Oriented Concepts, Databases and Applications - ACM Press, Addison-Wesley - 1989.

[Kaf 90] Kafura D. and Lee K.H. - ACT++: Building a Concurrent C++ with Actors - Journal of Object-Oriented Programming - May / June 1990.

[Lan 91] Lanusse A. and Roux P. - Actor Kernel Implementation - CEA / DEIN Report 91/47 - 1991.

[Lis 88] Liskov B. H. - Distributed Programming in ARGUS - Communications of the ACM, Vol. 31, No. 3 - March 1988.

[Mey 88] Meyer B. - Object-Oriented Software Construction - ed. Prentice-Hall - 1988.

[Nier 87] Nierstrasz O.M. - Active Objects in Hybrid - ACM SIGPLAN Notices, Proceedings OOPSLA 87, Vol. 22, No. 12 - December 1987.

[Pap 89] Papathomas M. - Concurrency Issues in Object-Oriented Programming Languages - in Object Oriented Development - ed. D. C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva - July 1989.

[Sun 90] Sun Microsystems - The Sun Lightweight Processes Library - 1990.

[Weg 87] Wegner P. - Dimensions of Object-Based Language Design - ACM SIGPLAN Notices, Proceedings OOPSLA 87, Vol. 22, No. 12 - December 1987.

[Yon 90] Yonezawa A. - ABCL: an Object-Oriented Concurrent System - The MIT Press - 1990.

Issues in Run-Time Support for Tightly-Coupled Parallel Processing

Dror G. Feitelson[*†] Yosi Ben-Asher[*] Moshe Ben Ezra
Iaakov Exman Lior Picherski Larry Rudolph[‡] Dror Zernik[§]

Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
E-mail contact: rudolph@cs.huji.ac.il

Abstract

This paper describes the novel aspects of the run-time implementation of a parallel programming language on a shared memory, bus-based multiprocessor. The language contains block-oriented parallel constructs and the implementation is built on top of a commercial real-time operating system kernel. The language/system interface includes functions that deal with groups of parallel activities as a whole, so as not to impose unnecessary serialization on the language implementation. Algorithms and data structures for the distribution of newly spawned activities, for the destruction of activities via the parallel analogues of the `break` and `return` instructions, and for synchronization with deadlock detection are described. Performance measurements are also given.

1 Introduction

The gap between the operating system level and the programming language level is filled by the runtime system. In parallel machines, the runtime system often plays a major role in the mapping and microscheduling of the parallel activities of the program onto the tasks or processes of the operating system in addition to being the direct interface between the two.

At the programming level, the most straightforward parallel construct is the `cobegin/coend` construct. This construct is widely used in the description of parallel algorithms, and has been included explicitly in a number of parallel programming languages, such as Occam [14] and *ParC* [19, 3]. Most multiprocessor run-time systems, however, do not support a system call that enables a number of activities to be created at once. For example, the Dynix system for Sequent machines only supports a Unix-like `fork` system call, and it is recommended that process creation and termination not be used to structure the program [17]. The Mach system divides this functionality into the ability to fork tasks and threads separately, but still only one at a time [25].

[*]Supported by an Eshkol fellowship from the Ministry of Science and Technology.
[†]Current address: IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598
[‡]Supported by a Kerin Amet grant.
[§]Current address: Department of Electrical Engineering, The Technion, 32000 Haifa, Israel.

At the system level, the program run-time environment is presented either with the physical machine, i.e. one task executing on each CPU, or with a virtual machine, i.e. a set of tasks (processes) that is managed by the operating system. We have chosen to build our system on top of a real-time operating system kernel. The run-time system should have more control over tasks and other system resources than is traditionally permitted in multi-user, interactive, time-sharing operating systems such as Unix. Instead of modifying such an operating system kernel, a real-time system kernel provides essentially the same control and requires significantly less manpower.

The goal of this paper is to discuss the issues that arise when system calls relating to groups of activities are included in the run-time system interface, and to describe the implementation of such system calls in MAXI, the Makbilan run-time system. Although MAXI is tailored for our specific parallel processor, the Makbilan, and for programs written in our parallel programming language, *ParC*, it contains techniques that are more widely applicable.

The basic structure of our work is similar to implementations of other parallel languages, e.g. the implementation of Concurrent *C* on a shared-memory multiprocessor [6]. However, our implementation has many distinctive and innovative features. The ones that are described in this paper include:

- *Dynamic process creation via global queue and multiprogramming via local queues:* there is no assumption made on the amount of computation required for parallel activities; it can range from a single instruction to many hundreds of thousands. We use of a global queue for balanced distribution of the work when new activities are spawned, but then use local queues for efficient scheduling of activities that have been created already.

- *Envelopes:* since activities may be numerous and very short, it is not desirable to always create a task for each activity. We introduce the notion of envelopes and a scheme for their creation and termination. An envelope is a task that is controlled by the low-level kernel and in which activities are executed. One envelope may execute many short-lived activities. On the other hand, if the activities of a parallel program are all heavy and long-lived, then the number of envelopes will become equal to the number of activities and their scheduling will be managed by the real-time kernel.

- *Parallel* `break` *and* `return` *instructions:* the closed parallel constructs of the programming language require new, nonobvious implementations to support the ability for one activity to terminate whole groups of related activities. In *ParC*, a `pbreak` or `preturn` operation can be executed in the middle of a nested parallel construct thereby terminating the entire construct. The participating activities are identified either by checking the structure of the activity tree or by using a special coding scheme.

- *Deadlock detection for busy waiting:* to keep the control over activities light-weight, synchronizations are preferably implemented by busy waiting. Most deadlock detection schemes will not work in this situation. A two-phase deadlock detection algorithm was developed; during normal execution it just does some local bookkeeping, but when it suspects a deadlock situation it performs a global check to ascertain that no waiting activities are about to resume execution.

The next section provides the necessary background about the *ParC* language, the Makbilan parallel processor, and the MAXI runtime system. Sections 3 and 4 present the

MAXI implementation of activity creation and forced termination, respectively. Section 5 discusses the implementation of synchronization primitives and deadlock detection.

2 Background

The *ParC* Language

ParC is a superset of the *C* programming language intended to support parallel programming in a shared memory environment [19, 3]. The main extensions to the *C* language are two block-oriented parallel language constructs, `parblock` and `parfor`; the first indicates that the constituent sub-blocks execute in parallel, while the second indicates that iterations of the loop body be done in parallel. Each sub-block or iteration is called an *activity*. These constructs may be nested in arbitrary ways, creating a tree of activities where all the leaves are executing in parallel. `pbreak` and `preturn` statements may be used to exit these constructs prematurely, in analogy with `break` and `return`. This forces the parallel activities created by them to terminate.

The *ParC* memory model includes both shared and private data structures. The accessibility of the data structures is determined by static scoping rules. Thus a variable that is declared within the block of code that defines a certain activity is accessible only by that activity and its descendants. Global variables are shared by all the activities. All this is implemented by the *ParC* compiler, which sets up pointers between the stacks of the different activities [3]. Therefore no run-time support is needed.

In addition to the parallel constructs, there are three main synchronization mechanisms. The first is a fetch-and-add instruction, denoted `faa`, that provides an atomic "add to memory" operation. The second is the `sync` instruction, which implements a barrier synchronization among all the activities created by a certain parallel construct. This instruction would typically appear in all of these activities, or else activities in which it does appear are forced to wait for the termination of activities in which it does not. The third synchronization mechanism is semaphores.

The MAXI System

MAXI is an acronym for the **Ma**kbilan **S**ystem, based on an abuse of the English alphabet. The system is partitioned into two main layers: a run time library that supports the *ParC* constructs, and a local kernel on each board (Figure 1). The run time library is the subject of this paper; in the sequel, the term MAXI refers to this library. The local kernel is Intel's RMK [15], which is a real-time kernel designed to use hardware support provided by the 386 and the Multibus-II (which are components of our hardware base). This kernel is highly optimized to provide fast task creation, termination, and context switching. In fact, context switching in RMK is faster than the context switching instruction provided in the 386 instruction set, because the kernel manipulates the hardware data structures directly [15, p. 7-35]: we clocked it at $72\mu s$ [2]. Parallel activities are implemented by RMK tasks (which are the RMK equivalents of Unix processes). The overhead for task creation and termination is about 1ms [2]. The scheduling time quantum is 50 ms.

Note that RMK is not an inherent part of the system: alternative kernels that support local task manipulation can be used. The run-time library code is fairly portable, because all machine specific details are hidden in the underlying kernel. Using a real-time kernel like RMK is better than a Unix kernel, for example, because it is more efficient and allows the

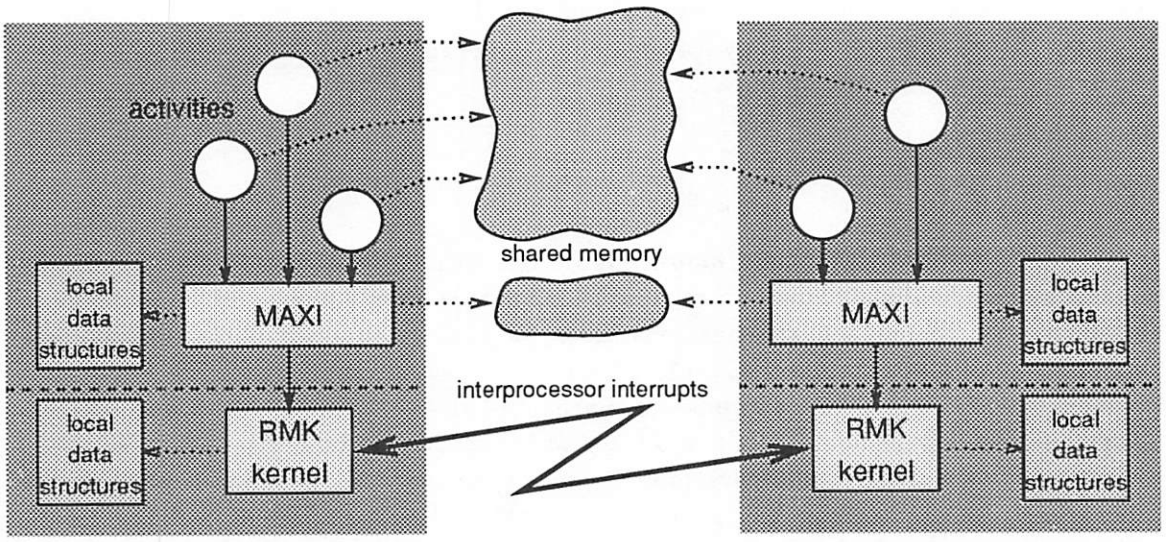


Figure 1: *Interactions among components of the application and MAXI.*

application more involvement in the system state. Thus task priorities, memory mapping, and stack allocation can be controlled, and a high-level interface to the interprocessor interrupts facility is available.

ParC and MAXI are intended to provide a general purpose convenient environment for users. The current version only supports a single user at a time, but a multiuser version is planned for the future. A basic guideline in the system design is the desire to shift some of the burden of parallel programming from the user to the system. Thus user programs enjoy the full benefits of a shared-memory environment. However, they are not allowed full control over the processors and memory mapping. Instead, the programs issue high-level *ParC* instructions and leave the implementation details to the system (top of Figure 1).

The MAXI design emphasizes asynchronous distributed operation without unnecessary interdependencies between processors. The system usually does not load the resources for parallel interactions, leaving them free for the user programs. Each processor has a local copy of the run-time library and kernel, complete with local data structures (Figure 1). MAXI is still a parallel program, but it only uses global data structures in shared memory for relatively infrequent events, such as the spawning of new activities.

The lowest level, the RMK kernel, is completely distributed. The kernels are local and do not know of each other. However, they provide an interprocessor interrupt facility, which allows the MAXI system on one board to cause all the other boards to perform some action in unison. The main use of this facility is to `pbreak` out of a parallel construct or to `exit` the program altogether.

The Makbilan Testbed

The Makbilan research multiprocessor consists of up to 16 processor boards in a Multibus-II cage. Each board has an Intel 386 processor running at 20 MHz, providing about 4 MIPS. It also has a 387 mathematical co-processor, a message passing co-processor, and 4MB of memory. Memory on remote boards may be accessed through the bus, thus supporting a shared-memory model. As access to on-board memory is faster than access to memory on remote boards, Makbilan is a non-uniform memory access (NUMA) machine [2]. The processors have on-board caches, but they do not cache remote references. Hence there is no issue of cache coherence.

The box also includes one board that acts as a Unix host, a bus controller, a peripherals interface, and a terminal controller. Users log on to the Unix board, and can then load and execute *ParC* programs on all the other boards.

3 Support for Parallel Blocks

The semantics of parallel blocks and loops imply that the constituent activities be spread out and executed in parallel on different processors, while the parent activity waits. The parent resumes its execution only when all the child activities terminate. Thus the main run-time support required for such constructs is the distribution and execution of the activities, as well as reporting back upon their termination. We initially leave the execution, i.e. the actual creation of activities and their scheduling, to the local kernel. Later we discuss how this too can be optimized.

The issue of work distribution links the scheduling schemes with the mapping strategies [12]. One extreme possibility is to use self-scheduling, where all the ready activities are kept in a global pool and idle processors take work from the pool [8, 18]. In this case the mapping of activities to processors is not preserved, but rather changes at each context switch. The other extreme is static, program controlled mapping, where each processor does local scheduling of the activities mapped to it [22].

MAXI leans towards the second approach in that there is a static mapping of activities. However, the mapping happens at run-time and depends on the system state. Static mapping is chosen for several reasons. At the system level, local queues are considered more efficient because activities do not move from one board to another. Thus state information such as the activity's stack is always local and does not have to be copied. Local queues also reduce contention for global locks, which could degrade performance [1]. At the application level, the fact that activities do not migrate also opens the possibility for associating data structures with activities, so that these data structures are always in local memory (this is important as we are dealing with a NUMA architecture). For example, we can create a shared array such that different parts of the array are local to different activities executing on distinct processors. All these considerations mesh nicely with the fact that RMK is based on local queues, and add impetus to its use. Had we decided to use a global queue we would have been unable to use RMK, as we do not have access to the data structures used by the RMK kernel to implement its scheduling. Thus it would have been difficult to force independent kernels on distinct boards to use the same run queue.

Spreading the Work

As we do not use run-time migration for load balancing, it is important to balance the initial mapping of activities to processors. The MAXI implementation achieves load balancing in a straightforward manner, by using a global list of spawned activities, from which all the processors take additional work. Each processor has a special activity that belongs to the run-time system to do this; it is called `get_work`. Whenever it is scheduled, it takes one new activity from the global list. As the `get_work` activity competes with other activities for the processor, the rate at which it is scheduled depends on the load on the processor.

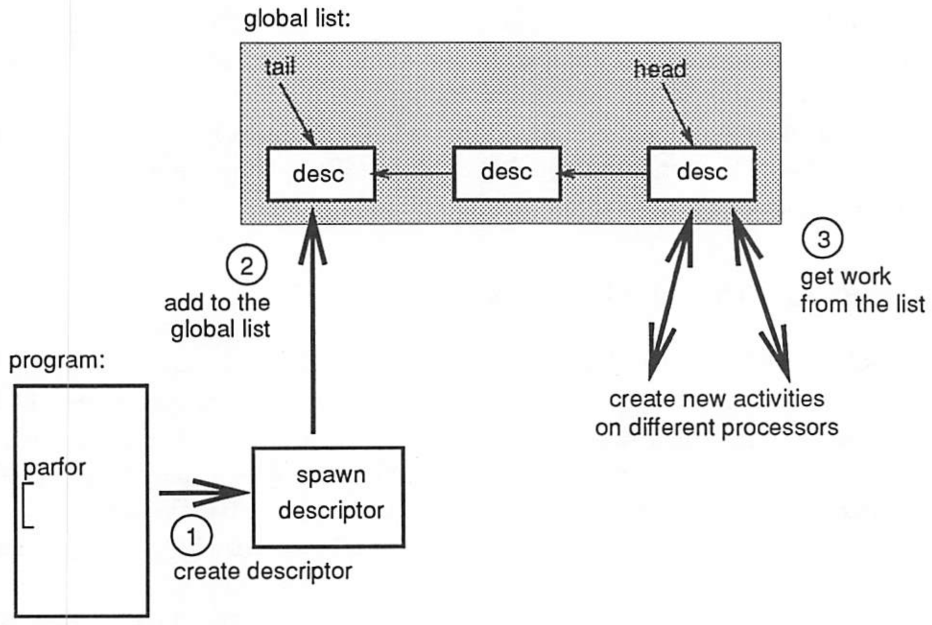


Figure 2: *Implementing spawns by a global list of descriptors.*

Thus it will be scheduled more often on lightly loaded processors, causing them to take more additional work [20].

Each item on the global list is a data structure called a *spawn descriptor* (Figure 2). It specifies the number of activities that should be spawned, the code that they should execute, and possibly arguments that should be passed to them. The descriptor also includes a counter of the number of activities that have terminated, additional data structures used in the implementation of synchronization primitives, and a pointer to the parent activity (this is used to resume the parent when the last activity terminates). Whenever a parallel construct is executed, a new spawn descriptor is added to the tail of the list. New activities are generated from the descriptor at the head of the list. The head and tail are locked independently, allowing spawns and activity creation to proceed in parallel. As activities are always generated from the oldest descriptor, the activity tree is executed in breadth-first order. This ensures a degree of fairness in the execution [3].

The effectiveness of the load balancing scheme is demonstrated by our measurement results from the following experiment. A total work of 100,000,000 assignments to a global variable was divided equally among different numbers of activities, which were executed on a 10-processor configuration. The results are shown in Figure 3. For each number of activities used, this shows the total time required and a histogram of the way the activities came to be distributed among the processors. The important point to notice is that balancing through the shared list results in better performance than just dishing out equal portions of the work to all the processors.

When there were only 100 activities, each doing 1,000,000 assignments, the load was divided equally: each processor executed exactly 10 activities. This can be seen from the leftmost bar for each PE — they are all the same height. However, the total execution time was about 100 seconds, which was sub-optimal, as witnessed by the fact that when there were a few thousand activities the execution time was reduced by 15%, requiring about 85 seconds (the minimum occurs for about 1900 activities). The distribution of the work shows

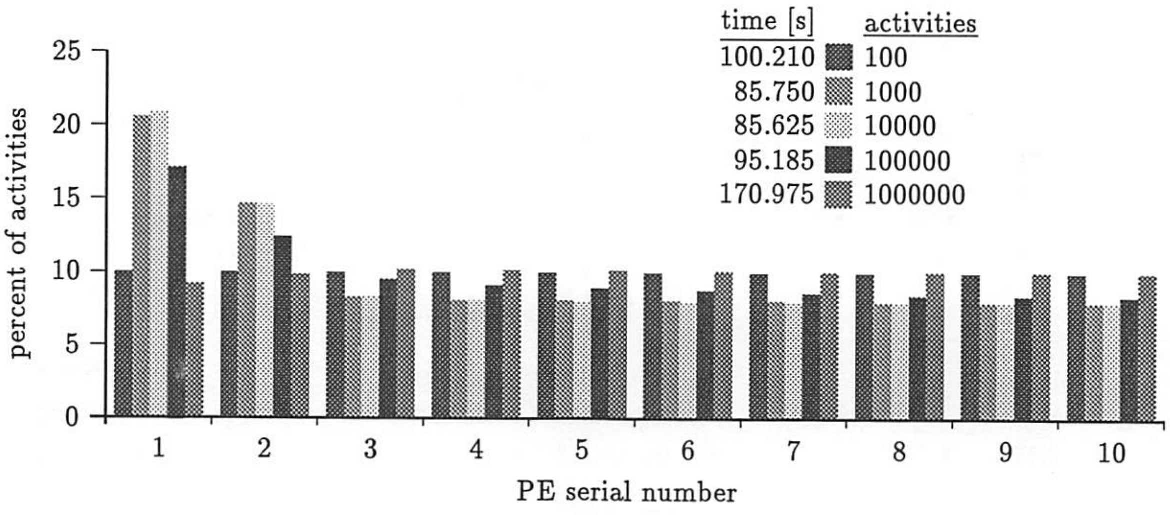


Figure 3: *Results of load balancing experiment. The time is the parallel completion time. It is therefore best divide 100,000,000 operations into several thousand activities when 10 processors are used.*

that this reduction was the result of non-equal loads: processor #1 executed about $2\frac{1}{2}$ times as many activities as processors 3 through 10. The reason for this strange behavior is that processor #1 is more efficient, because both the spawn descriptor and the global variable happen to be in its local memory. When the pool of activities is big enough, processor #1 takes more of them, executes them faster, and thus reduces the total execution time. When there are only a few activities, processor #1 might finish his earlier then other processors, but then it is left with nothing useful to do because all the other activities have been picked up already. Processor #2 is more efficient than the others because it has a higher bus priority.

When the number of activities was very large (1,000,000 activities each doing only 100 assignments), the situation is reversed. In this case the contention for the spawn descriptor and the overhead for spawning activities are the dominant factors, therefore the performance deteriorates dramatically. In particular, processor #1 now manages a bit less work than the others, because the contention for the spawn descriptor loads its local memory. Indeed, the main drawback of the global list is that it has to be locked when activities are added or deleted from it, thus serializing the process of activity creation to some degree. Note, however, that the serialization is not complete because allocating local data structures for the activity can be done in parallel. Moreover, this drawback is well worth it because the global list allows the processors to balance their loads based on their relative performance, rather than just based on the number of activities that each executes.

Although some of the anomalous behavior is a result of the lack of any truly shared, non-local memory modules, similar peculiarities can be found on many parallel machines. All that is needed is for some processor to have slightly different performance characteristics, e.g. an attached I/O device or specially allocated OS function.

Support for Multiple Fine-Grain Activities

It is often convenient to express a parallel algorithm as being composed of a very large number of fine-grain parallel activities. Much research has been done on the mapping

```
get_work:
forever {
   1. if (global list not empty)
         create an envelope task
   2. yield processor
}
```

```
envelope:
while (global list not empty) {
   1. reset timer
   2. execute one activity
   3. if (last in spawn descriptor)
         report to parent
}
delete the envelope task
```

Figure 4: *High level description of the* get_work *task and of an envelope. Each processor has one* get_work *task in its local queue, and it creates additional envelopes as necessary.*

and scheduling of such fine-grain activities so as to achieve high performance. Dataflow architectures use extensive hardware support to achieve this goal [7, 24]. The common approach on conventional multiprocessors is to increase the granularity at compile time based on an analysis of the program structure (see, e.g., [21]). We advocate the support of ultra-light-weight tasks in the run-time system. We contend that efficient support is possible at run time without previous knowledge about the program. In fact, we hope to relieve the programmer from the need to investigate how to partition the work into activities in order to achieve good performance.

The naive approach to activity generation maps each activity to an RMK task. When there are very many independent and fine-grain activities, this approach suffers from the overhead involved in creating these tasks. In addition, the system must maintain large data structures to cope with all the tasks. The suggested optimization is based on the possibility that a single RMK task execute multiple activities, saving overhead and data structures. A new RMK task is not created for every activity, but rather only if an activity suspends or executes for longer than a scheduling time quantum. In effect, the granularity is increased on-line when possible.

A similar approach has also been suggested for functional programming [16]. That work enjoys the benefits of freedom from side-effects, which makes all the tasks independent and therefore makes it possible to combine them together in any way. Our mechanism is more general and flexible, and correctly handles situations in which one activity depends on another.

The idea is as follows. The get_work task on each processor looks at the global list as before, and creates a new task if it finds work (Figure 4). However, the created task does not execute a single activity; rather, it executes what we call *envelope* code. The envelope executes activities in a serial loop, with one activity executed in each iteration (Figure 4). Whenever an activity terminates, it goes back to the global list to get a new one. Only when the global list is empty does the envelope delete itself. The timer is reset to a full time quantum before each activity is started. Thus if the activities are indeed independent and fine-grained, the envelope will not be preempted. Therefore there is a good chance that one RMK task per processor will execute all of them, without the overhead for generating additional RMK tasks and without the overhead of context switching. If, however, an activity suspends execution due to synchronization constraints, or executes for longer than a time quantum, the envelope *is* preempted. A new envelope is then created by get_work to deal with the rest of the activities. In the extreme case, this degenerates to the naive

code	*time [s]*		*improvement*
	default	*envelopes*	*factor*

`parfor 100,000`
`    g = 1;`
	20.245	7.265	2.787

`lparfor 100,000`
`    g = 1;`
	0.165	0.165	1.000

`parfor 1,000`
`    sync;`
	4.960	4.970	0.998

`parfor 100`
`    parfor 100`
`        sync;`
	7.495	1.630	4.598

Table I: *Experimental results using envelopes. A five-processor configuration was used.*

algorithm where a separate RMK task was created for each activity at the outset.

Note that envelopes are more flexible than previous proposals for pre-creation of tasks (e.g. in [4]), because the number generated is adapted to the characteristics of the program. If it is possible to save, we save, but if independent tasks are needed, they are created. We may miss when independent tasks are very long, but then the overhead is small relative to the total execution time, so the envelopes are still competitive with an optimal execution that would not create redundant tasks. All this does not require any user intervention.

It should also be noted, however, that envelopes are less flexible than the thread management mechanisms used in various thread packages (e.g. [10, 26]). These mechanisms use their knowledge of the application to perform fine-grain thread scheduling without operating system intervention. In effect, they replicate operating system services within the application. We feel this type of implementation is overly complex, especially since the RMK kernel we are using is rather efficient in its own right.

Experimental results using a preliminary version[1] of envelopes are presented in Table I. Four different codes are used as examples (the code given in the table is shorthand that captures the essence of the real code, but is not real *ParC* syntax). In the first, 100,000 activities are spawned, and each performs a single global assignment. Using envelopes is nearly 2.8 times faster. The second is identical, except for using the `lparfor` construct. This construct tells the compiler that the spawned activities are independent, and that they should be chunked into P equal chunks, one per processor. Thus the compiler substitutes the user directive to spawn 100,000 activities by a directive to spawn just P activities, and in addition generates code by which these activities each perform $100,000/P$ of the original assignments. For the very fine grain activities in the example, this is very efficient; using envelopes doesn't change anything because there is actually only one RMK task on each processor.

The third example concerns 1000 activities that perform a barrier synchronization. In order to do so all of the activities have to be spawned, so the envelopes gain no advantage. However, if there are multiple sets of synchronizing activities, as in the last example, envelopes can be reused for different activities. This leads to a speedup of nearly 4.6.

To further reduce the overhead, there should be no interprocessor interaction involved in the execution of independent fine-grain activities. In particular, possible contention for

[1]The main difference from the above description is that the timer is not reset for each activity that is executed.

the global list should be avoided. This can be achieved by partitioning the activities into disjoint sets, which are associated with the different processors. The relationship between the different components of the activity execution mechanism is then as follows:

- When a set of activities should be spawned, a spawn descriptor is placed in the global list as before.
- When the get_work task on any processor finds a spawn descriptor in the global list, it creates a local *representative* of the descriptor. A certain fraction of the activities are moved from the spawn descriptor to the representative. All local representatives are linked together, creating a local list.
- Envelopes loop and take activities for execution from the local representatives. As only one envelope can execute at a time on a certain processor, no locks are needed.

The main question that remains is how to divide the activities between the representatives. Equal division may not be optimal if the activities are not identical. A promising approach is to always allocate $1/P$ of the remaining activities, as was suggested for guided self-scheduling of parallel loop iterations [18].

The original implementation of MAXI used the naive approach, with a separate RMK task for each activity. A run time library based on envelopes has also been written and used for the experiment described above, and additional versions with more sophisticated features such as representatives are now being implemented (see [23] for details). We ultimately hope to achieve performance similar to that of an lparfor construct, without having to use it explicitly.

4 Forced Termination

Parallel activities in *ParC* programs are defined implicitly by blocks of code in parallel constructs. They are created and terminated dynamically at run time. There is no equivalent to the Unix fork style, which tells the system to "create an additional activity". Consequently, activities do not have an ID at the language level. Moreover, there is no way to kill an activity or send it a signal. However, it is possible to terminate all the activities that were generated by the same construct, together with any descendants created by nested constructs. This happens when one of the activities tries to jump out of the construct, by issuing a pbreak or preturn instruction. For example, such behavior is useful when a set of parallel activities are spawned to speed up a search through a large data structure. The activity that finds the required item then pbreaks, terminating the search of all the other activities [3].

Terminating a subtree of activities can be done in synchronous or asynchronous styles. The synchronous style means that the whole system is interrupted and immediately takes steps to terminate the relevant activities. The asynchronous style means that a notice about terminating the subtree is posted in shared memory, and each processor does its part at its convenience. We advocate the synchronous style, because the whole point of the pbreak and preturn statements is to stop various activities from doing spurious work. It is therefore imperative that the subtree be deleted as soon as possible. This also prevents situations where new activities are generated at a higher rate than existing ones are terminated.

pbreak and preturn in parallel constructs are implemented by scanning all the activities in the system and deleting those belonging to the subtree that is being terminated. The main challenge is to identify the required activities. This is complicated by the fact that the activities in each level of the subtree are spread across all the processors, and the only link between an activity and its grandparent that happen to be on the same processor may be a third activity on another processor. Implementing preturn is further complicated by the fact that the root of the subtree is not necessarily the direct parent of the activity that performs the preturn, and it is also necessary to pass a return value. However, these problems are easily dealt with by a source-level transformation, which is implemented by the *ParC* compiler [9].

We have designed three algorithms for identifying the related activities that should be terminated. The considerations in comparing them involve the overhead that is required when activities are spawned, and the resulting efficiency of terminating a subtree. If one assumes that forced termination is rare, it is better to reduce the overhead of spawning at the expense of more costly forced termination. If, on the other hand, forced termination happens frequently, it should be optimized at the expense of more bookkeeping during spawning.

The first algorithm is to follow the recursive structure of the activity tree, deleting only the *direct* children of a given activity at each stage. To do so, it is enough to broadcast the parent activity's ID[2] to all the processors. If any of these children have additional descendants, a recursive broadcast is sent with the child's ID. This scheme is completely general, conceptually simple, and can deal with any tree structure. It also does not add any overhead when activities are spawned. However, it requires all the activities to be scanned again for each internal node in the subtree when the subtree is being terminated. This repeated scanning requires complicated coordination between the processors, to ensure that the whole subtree is indeed deleted and that the parent activity is not resumed too soon.

Alternatively, a bottom-up version of this approach may be used. Again, the activity ID of the root of the subtree that is being terminated is broadcast to all the processors. The activities on each processor are scanned, and the parent pointers are followed from each activity up to the root of the activity tree. If the subtree root is encountered on the way, the activity is deleted. The cost of this approach is proportional to the number of activities in the system multiplied by the depth of the whole activity tree. Its main drawback is that it links data structures in distinct processors, forcing many remote references to be made. There is also a danger that the links in the top levels of the tree would become hot-spots, and that contention for them would reduce system performance. However, this can be avoided by marking each activity that is identified as being either in or out of the subtree, and searching only until a marked activity is found rather than up to the root.

The third algorithm is to use a coding scheme that allows all the descendants of a given activity to be identified at once. For example, activities may be identified by *ranges* of (unsigned) integers, written as $[b, t]$. The first activity in the program is identified by the whole range from $b = 1$ to $t = 2^{32} - 1$. When an activity identified by $[b, t]$ spawns k children, the range is divided into k; the first child is then identified by $\left[b, b + \frac{t-b}{k}\right]$, the second by $\left[b + \frac{t-b}{k} + 1, b + \frac{2(t-b)}{k}\right]$, and so on. Activities in a subtree rooted at $[b, t]$ are then easy to identify: they have ranges that are a subrange of $[b, t]$.

This scheme is much simpler than the previous one, in the sense that it is easy to maintain the ranges and the search time is only proportional to the number of activities. It main drawback is that it limits the size of the activity tree that can be spawned. However,

[2]While activities do not have IDs at the language level, they do at the system level. This is a system-wide unique integer.

algorithm	*increase in spawn overhead*	*average operations per killed activity*
coding	6%	6.7–8.4
bottom-up	21%	12.5–13.5

Table II: *Comparison of the performance of two implementations of the kill-tree operation. A number of different trees with about 4000 activities were used.*

there is a nice tradeoff between the depth and width of the tree. With 32-bit words, for example, only 3 levels of nesting are possible if a thousand activities are spawned each time. But if only two are spawned, 32 levels of nesting are supported. If longwords are used for the range boundaries, these numbers become 6 and 64 levels respectively. Note that by using any integers for range bounds, rather than using bit positions, the restrictions on the activity tree are reduced. It is expected that in most cases system tables will be the limiting resource, and not the coding scheme.

Run-time libraries implementing the last two algorithms have been written [9]. The first algorithm was rejected because it was felt that the overhead involved in the repeated broadcasts and synchronization was too great. The performance of the two approaches that were implemented is tabulated in Table II. As expected, the coding scheme is more efficient. However, considering that the coding limits the structure of the activity tree, we decided to provide users with both options. Users who know that their program does not spawn too many activities may use the coding scheme. Otherwise, the more general bottom-up scheme should be used.

5 Synchronization and Deadlock Detection

Barrier synchronization and semaphores are synchronization primitives that might require an activity to suspend execution and wait for other activities to catch up. This can be done by suspending the RMK task that implements the activity, thus causing RMK to stop considering this task as eligible to run. Alternatively, busy waiting may be used. In order to avoid wasting processing resources, the busy waiting loop should contain an instruction to yield the processor if the awaited condition is not satisfied. This form of busy waiting is used in MAXI.

When activities are delayed, there is always the danger that they will be delayed indefinitely, leading to deadlock. In the context of parallel programs, an activity may wait for an event defined by another activity, as opposed to concurrent systems where a process only waits for resources held by competing processes. Thus classical techniques developed for deadlock prevention and detection in concurrent systems are irrelevant when dealing with parallel programs. Specifically, maintaining a wait-for graph and looking for cycles in it is impossible, because an activity may not know who it is waiting for. Likewise, the definition of deadlock cannot be based on cycles between activities. We therefore use an alternative definition, by which deadlock means that *all* the activities are stuck and cannot proceed for some reason. This immediately points out a deadlock detection algorithm: simply count how many activities are stuck.

The problem is that the transition from "stuck" to "unstuck" may not take effect for some time, leading to false alarms. For example, consider a set of activities that are all stuck at a barrier synchronization, except for one that is stuck waiting for its descendants to terminate. When the last child terminates it raises a flag in shared memory indicating that the parent should be resumed, and deletes itself. During the time until the flag is noticed, the system seems to be in deadlock because all the activities are marked as stuck. Some sort of *global* interaction is needed to prevent such scenarios.

A simple solution is to use global counters: the last child to terminate can decrement the global counter of stuck activities before it deletes itself. Thus it becomes globally known that some task (the parent) is actually unstuck, even if it was not explicitly resumed yet. However, contention for the global counters would soon cause them to become hot-spots, degrading system performance. The following distributed algorithm solves the deadlock detection problem with minimal interference between processors:

1. The run-time system on each processor maintains information on local activities, by counting the total number of activities and the number that are stuck. Note that this is completely local. It sets a global bit if all the activities are stuck and the processor has nothing to do.

2. The run-time system on each processor also counts its net contribution to the number of activities that become unstuck system wide. This counter is incremented whenever a local event *causes* some activity to become unstuck, and decremented whenever a local activity *notices* that it has become unstuck. Returning to the above example, the last child to terminate increments the counter, and when the parent actually resumes execution, it decrements the counter. Note that if they are on different processors, these are distinct local counters.

3. The processor that sets the last bit assumes that all activities on all processors are stuck, so it initiates a synchronized computation of global state. This is done by summing the local "unstuck" counters from all the processors. If the sum is zero, it means that all the events have been noticed, and no more activities are in the process of becoming unstuck. Therefore all the activities are indeed stuck, and deadlock is announced. If the sum is larger than zero, there is no deadlock.

This deadlock detection algorithm has been implemented in MAXI. It is used both to detect deadlocks in application programs, and to detect deadlocks due to exhaustion of system resources (e.g. when all existing tasks are waiting for tasks that cannot be spawned because the system tables are full). A correctness proof and the details of how it is used to implement barrier synchronization and semaphores may be found in [11].

6 Conclusions

Most multiprocessor operating systems are straightforward adaptations of uniprocessor systems. This forces parallel language implementations to be based on primitives that were originally designed for time-sharing systems, such as `fork` or `kill`. As a result unnecessary serialization may occur.

The MAXI run-time library that supports *ParC* programs is based on an interface that deals with groups of activities at once. Thus efficient parallel implementations are possible. This poses new challenges and opportunities in the area of parallel system design. Algorithms for the creation of groups of new activities, for the termination of subtrees of activities, and for synchronization with deadlock detection were developed in response to this challenge.

We now have several working versions of the MAXI run-time library which provide run time support for the features of the *ParC* language. Versions based on alternative algorithmic solutions are being used to conduct experiments and compare the various approaches. For example, one version supports gang scheduling rather than scheduling each activity individually, which turns out to be important for fine-grain applications [13]. Other versions provide special additional services, such as monitoring [5]. The system is in the meantime also being used for research on parallel algorithms and for parallel programming courses. Technical reports describing these and other advances may be obtained form the secretariat of the Department of Computer Science at the Hebrew University of Jerusalem.

Acknowledgments

The *ParC* language was first developed by Yosi Ben-Asher. Some extensions were later added by Izhar Matkevich and Dana Ron. The compiler and simulator were written by Yosi Ben-Asher, Marcelo Bilezker, and Itzik Nudler. The simulator's task system was ported to the Makbilan multiprocessor by Omri Mann and Coby Metzger. The new system, which is described in this paper, was developed and implemented by Moshe Ben-Ezra, Lior Picherski, and Dror Feitelson. Yair Friedman implemented the kill-tree operation, and Constantine Shteiman is working on the envelopes. Dror Zernik, Larry Rudolph, Yosi Ben-Asher, and Sharon Broude took part in the discussions of various design alternatives. Martin Land kept the hardware up and running. Larry Rudolph has been in charge of the whole project since its inception, and Iaakov Exman replaced him when on sabbatical.

References

[1] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "*The performance implications of thread management alternatives for shared-memory multiprocessors*". *IEEE Trans. Comput.* **38(12)**, pp. 1631–1644, Dec 1989.

[2] Y. Ben-Asher and D. G. Feitelson, *Performance and Overhead Measurements on the Makbilan.* Technical Report 91-5, Dept. Computer Science, The Hebrew University of Jerusalem, Oct 1991.

[3] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, "*ParC — an extension of C for shared memory parallel processing*". Oct 1990. Manuscript, Dept. Computer Science, The Hebrew University of Jerusalem. Submitted for publication.

[4] P. Brinch Hansen, "*The programming language Concurrent Pascal*". *IEEE Trans. Softw. Eng.* **1(2)**, pp. 199–207, Jun 1975.

[5] D. Citron, I. Exman, and D. Feitelson, *MKMONITOR - A Parallel Monitor, or What You See is What You Program.* Technical Report 91-19, Dept. Computer Science, The Hebrew University of Jerusalem, Dec 1991.

[6] R. F. Cmelik, N. H. Gehani, and W. D. Roome, "*Experience with multiple processor versions of Concurrent C*". *IEEE Trans. Softw. Eng.* **15(3)**, pp. 335–344, Mar 1989.

[7] J. B. Dennis, "*Data flow supercomputers*". *Computer*, pp. 48–56, Nov 1980.

[8] J. Edler, A. Gottlieb, and J. Lipkis, "*Considerations for massively parallel UNIX systems on the NYU Ultracomputer and IBM RP3*". In *EUUG (European UNIX system User Group) Autumn '86 Conf. Proc.*, pp. 383–403, Sep 1986. Another version appeared in *Winter USENIX Technical Conf.*, pp. 193–210, Jan 1986.

[9] I. Exman, D. G. Feitelson, and Y. I. Freidman, *How To Kill an Activity Tree.* Technical Report 91-20, Dept. Computer Science, The Hebrew University of Jerusalem, Dec 1991.

[10] J. E. Faust and H. M. Levy, "*The performance of an object-oriented threads package*". In *Object-Oriented Prog. Syst., Lang., & Appl. Conf. Proc.*, pp. 278–288, Oct 1990.

[11] D. G. Feitelson, "*Deadlock detection without wait-for graphs*". *Parallel Computing* **17(12)**, pp. 1377–1383, Dec 1991.

[12] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[13] D. G. Feitelson and L. Rudolph, *Performance Benefits of Gang Scheduling for Fine Grain Synchronization.* Technical Report 91-8, Dept. Computer Science, The Hebrew University of Jerusalem, Oct 1991.

[14] INMOS Ltd., *Occam Programming Manual.* Prentice-Hall, 1984.

[15] Intel Corporation, *iRMK I.2 Real-Time Kernel Reference Manual.* 1988. Order number 462660.

[16] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr., "*Lazy task creation: a technique for increasing the granularity of parallel programs*". *IEEE Trans. Parallel & Distributed Syst.* **2(3)**, pp. 264–280, Jul 1991.

[17] A. Osterhaug (ed.), *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, 2nd ed., 1989.

[18] C. D. Polychronopoulos and D. J. Kuck, "*Guided self scheduling: a practical scheduling scheme for parallel supercomputers*". *IEEE Trans. Comput.* **C-36(12)**, pp. 1425–1439, Dec 1987.

[19] L. Rudolph and Y. Ben-Asher, *The PARC System.* Technical Report CS-88-8, Leibniz Center for Research in Computer Science, Hebrew University of Jerusalem, Aug 1988.

[20] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, "*A simple load balancing scheme for task allocation in parallel machines*". In 3rd *Symp. Parallel Algorithms & Architectures*, pp. 237–245, Jul 1991.

[21] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[22] M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "*Design rationale for Psyche, a general-purpose multiprocessor operating system*". In *Intl. Conf. Parallel Processing*, vol. II, pp. 255–262, Aug 1988.

[23] K. Shteiman, D. Feitelson, L. Rudolph, and I. Exman, *Envelopes.* Technical Report 92-1, Dept. Computer Science, The Hebrew University of Jerusalem, Jan 1992.

[24] V. P. Srini, "*An architectural comparison of dataflow systems*". *Computer* **19(3)**, pp. 68–88, Mar 1986.

[25] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "*Mach threads and the Unix kernel: the battle for control*". In *Proc. Summer USENIX Conf.*, pp. 185–197, Jun 1987.

[26] A. Tucker and A. Gupta, "*Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*". In 12th *Symp. Operating Systems Principles*, pp. 159–166, Dec 1989.

Experiences Implementing the Mintabs System on a MasPar MP-1

David Y. Hollinden, Debra A. Hensgen, and Philip A. Wilsey
Dept. of Electrical and Computer Engineering
University of Cincinnati, Cincinnati, OH 45221-0030

Abstract

This paper describes discoveries and experiences that arose when writing and optimizing the control algorithm for the Mintabs system. New lessons learned about programming of SIMD computers are conveyed. The purpose of the Mintabs system, being developed at the University of Cincinnati, is to implement instruction parallelism on massively parallel SIMD computers. Previously, SIMD computers have been used only to exploit data parallelism. Mintabs permits noncommunicating MIMD execution by compiling programs into a RISC-style instruction set, and loading them as data into the local memory space of each distinct processing element (PE) on the SIMD computer. Instructions are interpreted in parallel across all of the PEs upon broadcasts received from the central control unit (CU). The algorithm that runs on the CU is called the control algorithm. The lessons that we learned in implementing the algorithm on a MasPar MP-1 allowed us to achieve a speedup of over 1,600 for non-branching programs, on a 8K processor machine. We successfully overlapped much of our computation with instruction loading, and we were able to modify the MINTABS instruction set to make it more useful.

1 Overview of the Mintabs System

This paper discusses experiences in developing a prototype for the Mintabs system. The Mintabs system is a new paradigm for obtaining speedup on SIMD computers. This prototype, developed on a MasPar MP-1 computer, demonstrates that the new paradigm is viable, and is also helping us to make major improvements in the paradigm.

Parallel processing is most commonly accomplished using one of two architectural styles: SIMD or MIMD. MIMD machines consist of a number of independent processors, each executing its own instruction stream. A problem is typically solved on a MIMD machine by assigning each processor its own unit of code and communicating results and data between processors.

By contrast, SIMD computers take advantage of data parallelism, rather than instruction parallelism. Several different styles of architectural design may be referred to as SIMD. One common example is the pipelining vector processor, but we restrict our discussion to massively parallel SIMD computers. The typical massively parallel SIMD computer consists of one control unit (CU) and a large array of processing elements (PEs), each with its own local data space. A single stream of instructions is executed. This stream executes either on the CU or is broadcast by the CU to the PE array so that each PE receives the same instruction simultaneously. When the CU broadcasts an instruction, the PEs act upon the instruction in unison.

SIMD programmers typically search for a way to map the data-parallel parts of their programs onto massively parallel SIMD machines by assigning each PE a part of the data, and broadcasting one set of instructions to all PEs. The PEs mask themselves into and out of the instruction stream based upon broadcast conditionals. This allows for if-then constructs that depend on each PE's local data. References describing these data parallel techniques abound [3, 4, 9].

Recently, work was begun at the University of Cincinnati on the Mintabs system [13]. The purpose of this system is to support a paradigm for exploiting instruction parallelism on massively parallel SIMD architectures. The new paradigm is based upon the fact that a program looks like data when considered as a low level abstraction. In the Mintabs system programs are compiled into a RISC-like instruction set called MINTABS. The resultant programs (MINTABS instructions and data) are then sent as data to the PE array, one program per processor. The CU then acts as an operating system or software architecture to execute the programs on the processing elements. Instead of executing each program's instruction stream, a control algorithm executes on the CU. This control algorithm causes each PE to execute one of its local instructions on its local data, and then causes each PE to increment its local program counter and update its pointers in lockstep. As each MINTABS instruction is broadcast by the CU, the PEs switch themselves into and out of the operation, depending upon the processing requirements of their local instruction. The experiences and discoveries that have resulted from implementing this control algorithm will be the focus of the body of this paper.

We believe that this unique approach to SIMD programming will be advantageous in several problem areas of parallel programming:

- The new paradigm is ideal for speeding applications that require a large number of noncommunicating processes. The Mothra programs used for augmentation of test data [1] are an example of a suitable application. While this is also the area in which MIMD machines perform best, we believe that the problems of interprocesses communication and synchronization can be best solved on massively parallel SIMD machines, since their implementation will be assisted by hardware already in place. The nature of massively parallel SIMD architectures provides other advantages for the new paradigm when contrasted with direct implementation on a MIMD machine. SIMD machines tend to have a great many more processors than MIMD machines, and are more easily expandable. Also, massively parallel SIMD is a relatively new technique, and rapid advances in hardware should occur as additional applications for these machines are discovered.

- Many applications do not exhibit massive data parallelism, although they do exhibit massive parallelism. One such example is the use of Monte Carlo methods for simulating the properties of individual particles as they are transmitted through a system [5]. Although such simulations exhibit considerable parallelism (each particle is modeled by a separate execution of the program), they do not map well onto SIMD processors [10]. The new paradigm will likely facilitate the parallelization of these applications.

- Some applications execute in distinct phases that are either data parallel or instruction parallel. With few exceptions [11], existing parallel processors do not support this type of execution, with the result that these applications are speeded only during certain phases of their execution, or they require extra hardware. The new paradigm promises to support efficient computation of combined SIMD/MIMD applications: during SIMD phases, the machine can be used in a traditional data-parallel manner; during MIMD phases, the new paradigm can be used.

We now describe the original instruction set of the MINTABS architecture. It consists of seven 3-address instructions with a common format. The format of all instructions is: 3-bit opcode, 6-bit addressing mode, 16-bit address of the first operand, 16-bit address of the second operand, 16-bit address of the third operand. Each operand may be addressed using immediate mode or direct or indirect addressing. The semantics of each instruction are given in the following table:

Instruction	**Semantics**
add(x, y, z)	MM[z] = MM[x] + MM[y]
and(x, y, z)	MM[z] = MM[x] **and** MM[y]
comp_add(x, y, z)	MM[z] = **not**(MM[x]) + MM[y]
shl(x, y, z)	MM[z] = MM[x] << MM[y]
shr(x, y, z)	MM[z] = MM[x] >> MM[y]
beq(x,y,z)	**if** MM[x] = MM[y] **then** PC = MM[z] **end if**
bal(x,y,z)	**if** MM[x] ≥ 0 **then** MM[z] = PC; PC = MM[y] **end if**

where PC denotes the local program counter. The MINTABS instruction set was chosen to produce a covering set of operations with the most commonly occurring operations being selected (or at least to keep the common operations fast). The number of instructions was kept small in order to maximize the number of participating processors. We demonstrate in the paper that on the MasPar MP-1, a different instruction set can be more useful in practice.

In order to show the viability of this approach, it is necessary to develop an efficient control algorithm. The remainder of this paper will discuss experiences and discoveries that arose from the authors' ongoing work in creating and optimizing the control algorithm for the Mintabs system.

2 The MasPar MP-1

This research required the use of a massively parallel SIMD machine. Massively parallel is typically defined as having at least 1000 processors available for computation. The typical configuration is one central control unit that broadcasts instructions to this large processor array. While many massively parallel SIMD machines exist, the MasPar MP-1 was chosen as the best suited machine for our purposes. The primary reasons included the size of the machine (up to 16K processors), and the relatively large amount of memory available for each processing element. The MasPar provides up to 64K bytes of local memory for each processor; by comparison, the Connection Machine CM-2 allows only 4K bits per processor, clearly unacceptable for holding even small programs. We will soon implement a system of paging, in order to allow larger programs to be run using the Mintabs system. We used a MasPar processor with 8K nodes for our prototype.

The MasPar MP-1 consists of a front-end machine, an I/O subsystem, and a data-parallel unit. The data-parallel unit consists of an array control unit (ACU), the large processor array, and interarray communication mechanisms. The ACU broadcasts operations to the processors from its 4-Gigabyte virtual paged instruction space. A processor either participates in the operation or is masked out of the operation, based upon a data-dependent condition code. Each processing element (PE) is a load/store arithmetic processing element with dedicated registers and RAM. The MP-1 can have up to 16K PEs, each with up to 64K bytes of local RAM and 40 32-bit local registers, 16 of which are general purpose and user accessible. The PE instruction set supports both direct and indirect addressing. Two means of interprocessor communication are available: a global router and an X-net. The X-net allows for fast communication between neighboring PEs.

The MasPar provides a high-level language, MPL, for programming the data processing unit. This language is based on C, and supports all features of the language as defined by Kernighan and Ritchie [6]. In addition, statements, keywords, and library functions have been added to support data parallel programming. Some of the more important additions are:

- A new keyword, `plural`, distinguishes between the addressing space on the ACU and on the individual PEs. Variables defined using the keyword plural are allocated identically on each PE in the PE array. Variables defined without this keyword (singular variables) are allocated on the ACU. In addition, pointers may be defined as singular pointers to plural data (one pointer exists on the ACU, but the data to which it points is on each of the PEs, in the same location), or as plural pointers to plural data (the pointers, as well as the data, are on each individual PE and may contain different values on different processors).
- Plural expressions are supported. All arithmetic and addressing operations defined in C are supported for plural data types.
- A new data type,`long long`, is defined as a 64-bit entity. Support is provided for 64-bit loads, stores, expressions and arithmetic.
- A library of routines is provided for data-parallel programming. Some of these routines load data uniformly across all PEs in the array, and others retrieve it. Additional routines provide data transfer and communications across the PE array.

The initial control algorithm was written entirely in MPL. As we progressed, it became necessary to generate and modify the resultant assembler code for reasons explained in the next section.

3 Initial Experiments

We ran some initial experiments to ascertain the viability of the Mintabs approach. Because of favorable results with these experiments, and knowledge gained from these studies, current work focuses on the implementation of a compiler to translate C to a slightly modified version of the MINTABS instruction set. This section describes our preliminary experimentation with Mintabs; the next section discusses discoveries and refinements to the Mintabs control algorithm that were necessary in mapping the Mintabs system to the MasPar MP-1.

3.1 Benchmark programs

In order to determine the viability of the Mintabs approach, it was first necessary to choose a method for measuring performance. We chose to use the ratio of the number of instructions executed per second sequentially to the number that were executed per second when running the programs in parallel. To calculate the sequential performance, we ran a large number of MINTABS instructions, uniformly distributed, on a single processing element on the MasPar MP-1. This test program showed that 51,840 MINTABS instructions are executed per second. The control algorithm under test was then run on the PE array, using programs randomly generated from uniform distributions of MINTABS instructions. The total number of instructions executed per second was computed. For 8K processors, 35,601,912 instructions per second were executed. The ratio of instructions per second for the parallel control algorithm to instructions per second for the sequential test program was defined to be the speedup for the initial timings. For 8K processors, the speedup was found to be 686.

In order to record timings for the control algorithms, some sort of source programs were needed as "test data". We decided to use random programs composed of uniform distributions of MINTABS instructions for this purpose. We had several motivations for this decision. First, we wanted a way to ascertain the goodness of the method, without first building a compiler. In addition, we wished to be as general as possible with our results. If any particular program was chosen as test data, the generality of the results could come into question – were they specifically optimized for those programs on which they were tested? Even so-called benchmark programs suffer from this shortcoming [2]. Finally, we gave some thought to the idea of varying the distribution of MINTABS instructions based upon distributions of instructions found in real applications. This is mentioned in more detail below.

3.2 The initial control algorithm

The initial implementation of the control algorithm uses the original MINTABS instruction set, and does not implement indirect addressing. The control algorithm first loads a MINTABS program into each PE in parallel using the MPL BlockIn command. It then generates a sequence of control signals so that each PE will be able to interpret its own instructions. Each MINTABS instruction is called out in turn. Based upon the value of their local current instruction, the PEs switch themselves into and out of the current operation. After each PE has executed its first instruction, the control algorithm commands each non-branching PE to increment its local PC and store its result in lockstep. This method of executing exactly one instruction on each PE and incrementing local variables in lockstep was found to be significantly faster than doing individual increments. The best timings we achieved with individual increments, without branching, resulted in a speedup of 424 for 8K processors; versions using lockstep increments achieved speedups of more than 700 without branch instructions. Another significant speedup discovered at this point was to use register operations whenever possible, since register accesses are about 10 times faster than memory accesses on the MasPar MP-1.

Branching proved to be a tricky problem for the random test programs, for a randomly generated branch could easily loop forever. For this reason, during generation of the random programs, backward branching was disallowed. Furthermore, in order to keep the timings as conservative as possible, each branch jumped ahead only one instruction, and data was chosen so that branches were always taken (taking the branch requires more execution time). When these branch instructions were added to the random test programs, we were able to achieve a speedup of 686.

4 Refinements and Discoveries

Having written a functional control algorithm, we next set out to improve it. In the process, we made several discoveries about the MasPar MP-1 that allowed us to optimize the control algorithm considerably. We also found it necessary to reexamine some of our original ideas regarding development of the control algorithm and the MINTABS instruction set.

4.1 Varying random MINTABS distributions

One idea that was considered as a possible refinement of the control algorithm was to vary the frequency of instructions called out by the control loop. The rationale behind this was that since some MINTABS instructions execute more often than others, perhaps greater progress could be made by executing these common instructions more often than the less common ones. To this end, we wrote a program that input a high-level distribution, such as the ones found by Tanenbaum [12], along with methods for implementation of these high-level constructs in the MINTABS instruction set. The output of this program was the distribution of MINTABS instructions corresponding to the original high-level distribution. We then modified the random MINTABS program generator to use any specified distribution, in addition to the uniform one. We generated a set of test programs based on a likely high level distribution, and ran them on the MasPar using the original control algorithm. This resulted in no change in the timing results gathered from the uniformly distributed random MINTABS instructions. The reason for this was simple; when timing a run of some 8K programs run in parallel, the final time will reflect the execution time of the slowest of the programs. Given that MINTABS has only 7 instructions, and that we were using 8K programs, at any given time some processor would want to execute a given instruction, however uncommon that instruction may have been. This was a lesson we saw again and again while writing these algorithms: always assume that at least one program somewhere will need to execute any given control path in the control algorithm.

4.2 Loads dominate timings

In order to determine how best to optimize the code, the performance of the control algorithm was hand-profiled. The resulting timings showed that the program was spending 52% of its time doing loads and stores, and only 30% actually executing MINTABS instructions. The remaining 18% was spent on initialization, loop tests, pointer manipulations, and Boolean tests. This led us to explore ways to use the loads to our advantage, especially since additional loading would be required to implement such features of the MINTABS instruction set as indirect addressing.

The MP-1 overlaps loads and stores with some register operations. The MasPar PEs can be viewed as a composition of an execution machine and a memory machine. The execution machine performs all data transformations and register operations, and the memory machine transfers data between memory, either on the ACU or on the PEs, and the PE registers. These two machines operate concurrently whenever possible. That is, requests to the memory machine (loads and stores) are queued and processed asynchronously while the ACU continues with non-conflicting register operations. This concurrency opened the door for several important optimizations.

4.3 Using prefetching

As mentioned above, the MasPar provides concurrency of load/stores and register operations, provided that there are no conflicts (contention for a register, for example). This, plus the fact that loads dominated our timings, led us to explore ways to maximize this concurrency. The result was to take our cue from the use of pipelining in RISC processors and rewrite the control algorithm using prefetching of instructions, addressing modes, and arguments. More precisely, let I_i denote the i^{th} instruction in a MINTABS instruction stream. Then in each iteration of the control algorithm, the following functions are performed:

1. Prefetch instruction I_{i+1} and its operands from the PE's local memory into the PE register set.
2. Execute instruction I_i using the previously fetched operands stored in PE registers and place the result in a PE register.
3. Store the results of the execution of instruction I_i back into local PE memory.

Thus the memory machine performs the first and third operations while the execution machine performs the second. This enables the MINTABS instruction interpretations to "hide" behind the loading on each cycle, greatly speeding execution.

4.3.1 Delayed branching

One of the disadvantages of implementing prefetching is the additional complexity it brings to branching. Something must be done with the prefetched next instruction when the current instruction takes a branch. We decided to use a delayed branching algorithm to solve this problem. When a branch occurrs, if it is taken, a Boolean flag is set in the loop. This prevents the registers from being updated with the prefetched instruction and operands. Meanwhile, the local PC is set to reflect the branch. An extra cycle is taken through the algorithm, in which no interpretation is done, but the prefetch is accomplished using the new value of the PC. In this way, branching is achieved with a minimal penalty to the branching PE and no cost to the non-branching processors. Another advantage of implementing delayed branching in the control algorithm is that compilers do not have to check for this condition and correct for it. Typical compiler-based delayed branching requires extra instructions, such as no-ops, to be executed; no extra instructions are executed using the method described above. Instead, some small number of processes skip an iteration while the others proceed.

4.3.2 Delayed loading

The use of prefetching, when combined with indirect addressing, results in a second problem. The address of instruction I_{i+1} is pre-fetched near the beginning of the control algorithm. If that address is actually the result of instruction I_i, its value will not be known until near the end of the control algorithm. We chose not to pre-fetch the operand for all processes near the bottom of the loop, since doing so would have reduced the possibility for overlap with register operations. We also chose not to pre-fetch near the bottom for only those processes affected by this data contention. As we found earlier, when a large number of processes are executing in parallel, at least one will need to take each available control path in each iteration of the control algorithm. Because at least one process will need this pre-fetch in each cycle, all processes would pay. Instead, we chose to delay loading the operand until the next iteration. This delayed loading is similar to the delayed branch, and both solutions have the same advantages: low cost (one throwaway cycle), and only the offending processor is affected.

4.4 A final problem

In our attempts to achieve parallelism with loads and register operations, we found some anomolous results. Specifically, we were not getting the overlap we expected, and in some cases we were getting no overlap at all! This led us to question whether the loads were in fact being queued and run in parallel with register operations. To determine what was actually happening, we ran the very simple program shown below. This program executes one load and one register instruction many times in a tight loop. The results for several instructions are summarized below, and are not too surprising: the short instructions are completely covered by the load, and the larger ones are at least partially covered.

```
Loop:

result = instructions[pc];      (Load instruction)
val1 op= val2;                  (Register operation)

where op = AND, ADD, MULtiply, SHiftLeft, or SHiftRight

Timings:

Load Alone: 15.92s

MUL  Alone: 10.71s          MUL + Load: 17.82s         Partial Overlap
ADD  Alone:  1.91s          ADD + Load: 15.92s         Total   Overlap
AND  Alone:  1.91s          AND + Load: 15.92s         Total   Overlap
```

4.4.1 The puzzle

Two instructions produced sharply anomolous results, however. The shift left and shift right instructions were highly suspect from the initial tests, since they seemed not to be overlapped at all. However, even given this expectation, the following timings came as a surprise:

```
Load Alone: 15.92s

SHL  Alone:  3.72s          SHL + Load: 24.63s         Worse than no Overlap
SHR  Alone:  3.62s          SHR + Load: 24.13s         Worse than no Overlap
```

How could the combined time be worse than the individual times added together? Was the shift not only not parallelizable, but actually slowing down the load?

4.4.2 The explanation

The answer came from an examination of the MasPar assembly code generated by the MPL compiler for the code fragment above. MasPar PE registers are bit addressable 32-bit registers. They are referenced as an offset from a base register (c0). Hence, register 31 would be written as 31*32[c0]. An immediate operand is prefixed with a dollar sign.

The load and shift instructions become the following assembly code:

```
/* Generated by load */

mov32   31*32[c0],0*32[c0]
shll32  $3,0*32[c0]
mov32   25*32[c0],1*32[c0]
add32   0*32[c0],1*32[c0]
ld64    *1*32[c0],28*32[c0]

/* Generated by shift left */

mov8    23*32[c0],0*32[c0]
shll32  0*32[c0],22*32[c0]
```

The answer is that shifts always block loads. When the ld64 is run by itself, it runs in parallel with some of the other instructions that comprise the MPL load instruction. When run with the MPL shift, however, the load blocks and cannot run in parallel with the other assembly instructions. Hence, the combined result is worse than the sum of the parts!

This answer was confirmed by MasPar: shift operations that specify shift lengths locally (on the PEs) will conflict with loads into PE memory, due to a conflict for the internal exponent register. These results led us to one more optimization: do away with the shifts! We substituted multiply and divide for the MINTABS shifts; these instructions are probably more useful anyway, and even though they take longer to execute than the shifts, they can be "hidden" by overlapping them with loads. We also substituted pointer references for bit-shifting when unpacking 64-bit entities. Finally, we generated the assembler code for the algorithm and replaced each shift left with the appropriate equivalent multiply command.

5 Conclusions and future work

Our work in implementing optimized control algorithms for the Mintabs system led us to a new version of MINTABS, and a new outlook on adding to this instruction set. We also were able to discover and take advantage of several unusual features of the MasPar MP-1.

5.1 New version of MINTABS

The original MINTABS instruction set was chosen to provide a simple, fast covering set of instructions so that the control algorithm could be as fast as possible. We considered including multiply and divide in MINTABS as well, and whether to extend it even further by including such things as floating point instructions. The research described above answered several of these questions. Multiply and divide were added to replace shift left and shift right; their ability to overlap with loads made this a clear win. In addition, programs written in this new instruction set should be smaller. Multiply translates into 23 non-multiply MINTABS instructions, and requires up to 263 instructions to execute. Similarly, divide translates into 39 non-divide MINTABS instructions, and can take up to 432 instructions to execute. The fact that loading still dominates the timings would indicate that the instruction set could be expanded even further with little performance penalty. We are currently implementing an interpreter for the MIPS R2000 instruction set. This instruction set consists of 74 instructions, and has the advantage of the existence of several fine optimizing compilers.

5.2 Future work

The Mintabs system is currently being tested for applications that generate a large number of executable programs that require little interprocess communication. We will then attempt to generalize to the simultaneous execution of programs that interrelate with each other, or of programs that have nothing to do with each other! A good amount of further research will be required to achieve that goal, however. Some of the more important points of interest are mentioned below.

5.2.1 Mutant programs

Current research using the Mintabs system concerns its use to speed a specific application: the execution of mutant programs for test data adequacy testing. Mutant program analysis was introduced by DeMillo [1] as a technique for determining test data adequacy in software engineering. A mutant program is generated from a program under test by applying a single instance of a mutant operator. These operators include statement deletion and replacement of relational operators. Each mutant program is then executed for each element in the data set, until either the mutant's output differs from the original program's output or the data set has been exhausted. If no differences in output are found, and if a mutant is not equivalent to the original program, the data set is augmented.

DeMillo's group built a system, Mothra [8], for generating mutant programs and executing them against data sets. Because the technique is computationally expensive, and because the mutant programs require almost no interprocess communication, this system was deemed an ideal candidate for the new paradigm. Previous attempts at SIMD parallelization for Mothra [7] were based on identification of basic blocks of code, and attempts to make the processes converge into basic blocks, which can then be executed in parallel. This method exhibited little speedup on a Connection Machine, due to divergence of the mutant programs. However, parallelism in the new paradigm is mostly independent of program divergence, and significant speedup is expected. A discussion of current progress in implementing mutant program execution using the Mintabs system may be found in [14] .

5.2.2 Other future work

Several other issues surrounding the Mintabs system will demand long term attention and research. A C to MINTABS compiler is currently under development, to facilitate the execution of more applications under the Mintabs system. Because application programs will typically become very large when compiled into the MINTABS instruction set, paging on the individual PEs is being studied.. Synchronization and communication between processes will become increasingly important as a larger base of application programs is considered for implementation on the Mintabs system. Transput is currently ignored due to the high overhead it requires; the initial implementation uses pre- and post-processing steps to process I/O. Although a good deal of research remains before the Mintabs system can be considered a general purpose system, the initial results have been extremely promising. We have seen speedup of about 700 for the initial control algorithm. A partially optimized version, without branching instructions, has shown speedup in excess of 1,600 for 8K processors. In this version, all of the computation has been overlapped by prefetched instruction loading. In addition, our speedups are linear with 2K, 4K, and 8K processors. This was an expected result. The overall execution time will become a constant once each MINTABS instruction is executed on at least one processor on every iteration of the control loop. After this plateau is reached, adding additional programs in parallel will add nothing to the overall execution time.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[3] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[4] R. M. Hord. *Parallel Supercomputing in SIMD Architectures.* CRC Press, Boca Raton, FL, 1990.

[5] C. Jacoboni and L. Reggiani. The monte carlo method for the solution of charge transport in semi conductors with applications to covalent materials. *Review of Modern Physics*, 55(3):645–705, July 1983.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[7] E. W. Krauser, A. P. Mathur, and V. Rego. High performance software testing on SIMD machines. Technical Report SERC-TR-17-P, Software Engineering Research Center, Purdue Univ., West Lafayette, IN, 1988.

[8] A. J. Offutt. *Automatic Test Data Generation.* PhD thesis, Ann Arbor, MI, 1988.

[9] D. Parkinson and J. Litt, editors. *Massively Parallel Computing with the DAP.* The MIT Press, Cambridge, MA, 1990.

[10] G. Rodrique, E. D. Giroux, and M. Pratt. Perspectives on large-scale scientific computation. *IEEE Computer*, 13(10):65–80, Oct. 1980.

[11] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, JR P. T. Mueller, JR H. E. Smalley, and S. D. Smith. Pasm: A partitionable SIMD/MIMD system for image processing and parallel recognition. *IEEE Trans. on Computers*, C-30(12):934–947, December 1981.

[12] A. S. Tanenbaum. Implications of structured programming for machine architecture. *Communications of the ACM*, 21(3):237–245, March 1979.

[13] P. A. Wilsey and D. A. Hensgen. Exploiting simd computers for general purpose computation. *Sixth International Parallel Processing Symposium*, 1992.

[14] P. A. Wilsey, D. A. Hensgen, C. E. Slusher, N. B. Abu-Ghazaeh, and D. Y. Hollinden. Exploiting SIMD computers for mutant program execution. *The 19th Annual International Symposium on Computer Architecture*, 1992. (submitted).

A An Implementation of the Control Algorithm

This algorithm shown below is an implementation of the central control algorithm for the MasPar massively parallel processor. The algorithm is written in a language called MPL. MPL is a C-like language with SIMD extensions developed by MasPar.

```
#include <fcntl.h>
#include <errno.h>
#include <mpl.h>
#include <sys/time.h>
#include "Boolean.h" /* define true and false */
#include "operation_ids.h" /* defines ADD, SUB, ..., HALT  and max program size */

#define MAXPROCESSORS 8192

typedef int data_t;

typedef struct prog{
int num_inst;
int num_data;
int PC;
long long instructions[MAXINSTRUCTIONS];
data_t data[MAXDATA];
}prog_t;

typedef struct prog_struct{
int num_progs;
prog_t program[MAXPROCESSORS];
}prog_struct_t;

visible do_it();

do_it(fe_prog_p)
prog_struct_t *fe_prog_p;
{
int num_programs;
/* Variables below allocated on each processor */

/* pointers to program, instructions and data parts */
plural prog_t my_prog;
register plural long long *plural inst_p;
register plural int *plural data_p;

/* to hold pc, instruction, operands and their modes */
/* pf_ means pre-fetched.  */
register plural short inst;
register plural short modes,pf_modes;
register plural int op1, op2, result,pf_op1, pf_op2;
register plural long long instr, pf_instr;
register plural int pc;

/*
  Flags:
Store   = Store the result of the current iteration
Dbranch = Delayed Branch
Dload   = Delayed Load
*/
register plural char Store,Dbranch,Dload;  /* flags */

copyIn(fe_prog_p, &(num_programs), sizeof(int));

if (iproc < num_programs) {

   /* Load the programs */

   blockIn(((int)fe_prog_p)+sizeof(int), &(my_prog), 0, 0, nxproc, nyproc,
sizeof(prog_t));

   inst_p = my_prog.instructions;
   data_p = my_prog.data;
   pc = 0;
   Store = true;
   Dbranch = false;
   Dload = false;

   /* Load the first instruction */

   instr = inst_p[pc];

   /* Unpack it */

   inst = (instr >> 24) & 0xff;
   modes = (instr >> 16) & 0xff;
   op2 =  (instr >> 48) & 0xffff;
   result = (instr >> 32) & 0xffff;

   /* Prefetch the first operands */

   if (modes & 0x30)
     op1 = data_p[op1];
   if (modes & 0x20)
     op1 = data_p[op1&0xf];
```

```
if (modes & 0x0c)
  op2 = data_p[op2];
if (modes & 0x08)
  op2 = data_p[op2&0xf];

/* Loop until all programs have finished */

while (inst - HALT) {

   /* Prefetch and unpack the next instruction */

   pf_instr = inst_p[pc + 1];
   pf_op1 =  pf_instr & 0xffff;
   pf_op2 =  (pf_instr >> 48) & 0xffff;
   pf_modes = (pf_instr >> 16) & 0xff;

   /* Check for data contention */

   if ((pf_op1 == result) || (pf_op2 == result)) {
     if (Dload) {

       /* Second load behind iteration: do not store result */

       Dload = false;
       Store = false;
     }
     else {

       /* First load behind iteration: store result */

       Dload = true;
       Store = true;
     }
   }

   /* Prefetch the next instruction's operands */

   if (pf_modes & 0x30)
     pf_op1 = data_p[pf_op1];
   if (pf_modes & 0x20)
     pf_op1 = data_p[pf_op1&0xf];
   if (pf_modes & 0x0c)
     pf_op2 = data_p[pf_op2];
   if (pf_modes & 0x08)
     pf_op2 = data_p[pf_op2&0xf];

   /* Perform the current instruction */
   /* Temporarily store result back into op1 for later lockstep store */
```

```
if (inst == ADD)
   op1 += op2;
if (inst == SUB)
   op1 -= op2;
if (inst == MUL)
   op1 *= op2;
if (inst == DIV)
   if (op2) op1 /= op2;
if (inst == AND)
   op1 &= op2;

if (inst == BEQ) {
   if (Dbranch) {

     /* Second delayed branch iteration, do not store result */

     Dbranch = false;
     Store = false;
   }
   else if (op1 == op2) {
     if (modes & 0x1)
       op1 = data_p[result] - 1;
     else if (modes & 0x2)
       op1 = data_p[data_p[result]&0xf] - 1;
     else
       op1 = result - 1;
     Dbranch = true;
     Store = false;
   }
}

if (inst == BAL) {
   if (Dbranch) {

     /* Second delayed branch iteration, do not store result */

     Dbranch = false;
     Store = false;
   }
   else if (op1 > 0) {
     op1 = pc;
     pc = op2 - 1;
     Dbranch = true;
   }
}

if (Store) {
```

```
          /* Store result temporarily placed in op1 */

          if (modes & 0x1)
            data_p[result] = op1;
          if (modes & 0x2)
            data_p[data_p[result]&0xf] = op1;
        }

        /* Do not update registers or increment pc if delayed branch/load */

        if ((!Dload) && (!Dbranch)) {
          ++pc;
          inst = (pf_instr >> 24) & 0xff;
          modes = pf_modes;
          op1 = pf_op1;
          op2 = pf_op2;
          result = (pf_instr >> 32) & 0xffff;
          Store = true;
        }
        }

     pc = 0;
     }
     blockOut(&my_prog, ((int)fe_prog_p)+sizeof(int),0, 0, nxproc, nyproc,
  sizeof(prog_t));

  }
  }
```

Experiences With Optimistic Synchronization For Distributed Operating Systems

Peter L. Reiher
Jet Propulsion Laboratory
reiher@onyx.jpl.nasa.gov

Abstract

Optimistic synchronization is a method of synchronizing parallel and distributed computations without the use of blocking. When non-optimistic systems would block, optimistic synchronization mechanisms permit operations to go ahead. If such optimism causes improper synchronization, the mis-synchronized work is undone and the entire system restored to a consistent state. This paper discusses the experiences of developing a distributed operating system based around optimistic synchronization, the Time Warp Operating System (TWOS). It covers the challenges of implementing such a system, the advantages of optimistic synchronization, and how well optimistic synchronization works in practice in TWOS, and offers advice for others developing systems using optimistic synchronization.

1. Introduction

Distributed systems, by their nature, require synchronization. The most common method of synchronization in distributed systems is some form of blocking. Whenever a processor needs to take some action that might cause synchronization problems with other processors, it blocks until it can be certain that no such problems will arise. There are many different ways to achieve synchronization using blocking, but all of them share the common property that certain actions must be delayed until the system is certain that proper synchronization is present.

An alternative, less used method of synchronization is optimistic synchronization. When a processor reaches a synchronization point, instead of blocking, the processor goes ahead and performs the action. In some cases, performing the action immediately is perfectly safe. If the processor had waited to obtain synchronization information, instead, it would have simply wasted its time until it was told to go ahead. In other cases, the processor would have been told that performing its action was not yet safe, and that the action would have to be delayed. In these cases, not waiting to obtain synchronization information causes an optimistically synchronized processor to make a synchronization error.

In the latter cases, optimistic synchronization solves the problem by restoring the state of the action to whatever it was before the synchronization point, in effect undoing all potentially erroneous work. Then the work can be redone in correct order. Care must be taken to ensure that all effects of performing work out of order are undone.

Whether optimistic synchronization will perform well depends on several factors. First, the probability that an action could be performed correctly without waiting for synchronization must be high. Second, the cost of waiting must be high. Third, the cost of undoing any mis-synchronized work must be low. In certain cases, the probability that an action can produce correct results even when done out of order can affect the performance of optimistic synchronization.

Optimistic synchronization has been used in a limited way in certain distributed database systems [Bhargava 82], [Carey 88]; and for some special purposes in operating systems. Lazy evaluation caching for name translation is one example of the latter; misdelivery of messages due to stale cache entries is effectively "rolled back" by obtaining fresh information and redelivering them. [Goldberg 92] investigated using optimistic methods to ensure consistency of replicated data. [Strom 90] discussed limited use of optimistic execution of processes in the Hermes system. However, until recently no operating systems that relied exclusively on optimistic synchronization have been developed.

The Time Warp Operating System (TWOS) is one such system. TWOS is a special purpose system designed to run discrete event simulations on parallel or distributed hardware with the goal of maximum speed. Certain of the characteristics of the discrete event simulation problem make it particularly suitable for optimistic synchronization, so TWOS was developed with optimistic synchronization from the start. TWOS has largely achieved its goal. In the process, much has been learned about optimistic synchronization. This paper describes some of the lessons learned and presents advice for others who are considering using optimistic synchronization in their systems.

2. The Time Warp Operating System

The Time Warp Operating System will be used as an example of a system using optimistic synchronization in this paper, so some familiarity with it is necessary. This section covers only those details necessary for this paper. A more complete discussion of TWOS can be found in [Jefferson 87].

TWOS is a special purpose operating system designed to run discrete event simulations at maximum speed on parallel or distributed hardware. It runs a single simulation at a time, devoting all available resources to the fastest possible completion of that simulation. TWOS simulations are decomposed into *objects*, which are spread across all available processors. ("Object" is an unfortunate choice of terminology, since TWOS objects are not the same as objects in the object-oriented programming sense, sharing just enough characteristics to cause confusion.) Objects communicate solely by messages, with no shared memory. Messages are timestamped with the simulation time at which they should be received, and the receipt of a message causes the receiving object to run an event at the receive simulation time. All simulation code is part of some event, except for certain initialization and termination procedures.

Objects never send messages into the past, so a proper sequential execution of the simulation would run all events in strict simulation time order. The distributed run of the simulation must produce results identical with those from a sequential run.

The parallel/distributed discrete event simulation problem can be solved with blocking synchronization [Chandy 79], but this solution has certain practical problems, including poor performance in some important cases [Fujimoto 90]. [Fujimoto 90] also contains a good survey of the large amount of work done in the field of parallel and distributed discrete event simulation.

TWOS is based on the theory of *virtual time* [Jefferson 82], which encompasses a complete optimistic synchronization mechanism. A virtual time system puts a time tag on every action in the system. These time tags are not necessarily directly related to real time in any way, but rather specify the order of actions in the application. For TWOS simulations, simulation time is mapped directly into virtual time.

In a TWOS run, each processor hosts several objects, scheduling them independently of all other processors. The TWOS scheduler always chooses the local object with an unprocessed event at the earliest simulation time to run next. Since TWOS' goal is fast completion of the overall simulation, fairness in scheduling objects is not a consideration. Therefore, TWOS does not employ time sliced or round robin scheduling. Objects are only pre-empted when another local object receives a message with an earlier timestamp than that of the event currently running.

Since each processor schedules without waiting for, or consulting with, other processors, at any given instant of real time the system's processors may be working at a wide range of virtual times. An object running at a low virtual time can send a message to another object at a higher virtual time. If the message is scheduled to arrive at a virtual time earlier than the receiving object is currently handling, the receiver must *roll back* his computation to the virtual receive time of the newly arrived message. Any erroneous work done by the out-of-order computation must be totally undone. Undoing the erroneous work requires throwing away local results and sending message cancellations to other objects. TWOS is able to correctly undo any work done prematurely, along with any side effects it may have had. TWOS rollback and message cancellation is totally transparent to the application program.

Every object has a set of private variables called its *state*, which cannot be directly examined by any other object. Every event causes the creation of a new version of the state, timestamped with the simulation time of the event. TWOS typically keeps multiple copies of each object's state in order to support rollback. Rollback causes any incorrect copies of the state to be deleted, and restores one of the earlier copies of the state.

Message cancellation in TWOS is performed by creating two copies of each message sent. One copy, called the *positive copy*, is sent to the receiving object. The other copy, called the *negative copy*, is retained by the sender. Should cancellation of a message be necessary, the negative copy is sent to the receiver. When the receiver tries to enqueue it, TWOS recognizes that two messages, identical except for their sign, are in the same queue. The message copies *annihilate*, and the receiver rolls back the event associated with the positive copy, if the event was already performed.

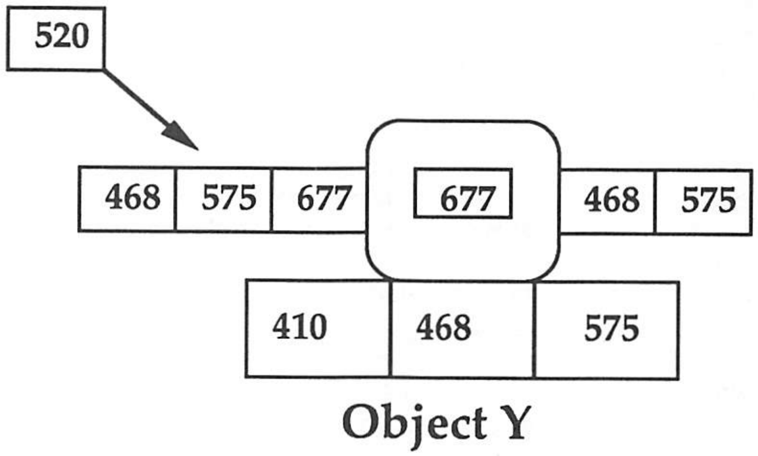


Figure 1: Object Y Is About To Roll Back

Figure 1 shows a TWOS object about to undergo a rollback. An object named Y has performed events for virtual times up to 677. In this diagram, Y consists of a queue of input messages that have been received and will cause events to be run; a queue of output messages produced by events run by this object; a queue of states produced by events run by this object; and a control block storing, among other things, the time of the next event to be run. Object Y is about to receive a new message, at time 520. But Y has already optimistically run an event for time 575, which should have been run after the event at time 520. Therefore, TWOS must roll back object Y to restore its state for the event at time 520 and undo any computation resulting from the incorrectly synchronized event at time 575. Figure 2 shows the local result of this rollback. Note, however, that Y sent a message to some other object at time 575, which could have caused that object to run further events and send more messages. Though not described here, TWOS is able to correctly roll back and cancel any of the secondary effects of an improperly synchronized event. See [Jefferson 87] for further details.

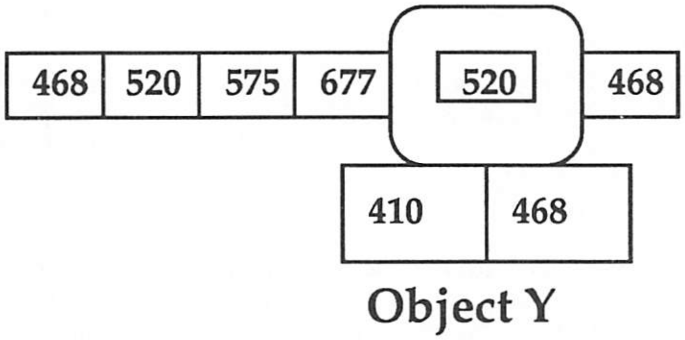


Figure 2: Object Y After the Rollback

At any given moment in a TWOS run, the simulation's objects have performed some work correctly, and some work in error. TWOS periodically calculates the earliest virtual time that could still be in error. Any work done for virtual times earlier than that time will never be rolled back, and can be *committed*. Both events and messages can be committed. A *committed message* is one that would have been sent in the sequential run of the program, and a *committed event* is one that would have been performed in the sequential run of the program. In essence, these committed actions represent the correct path of computation for a simulation. To meet its definition of correct behavior, any event or message TWOS commits must exactly correspond to an event or message that would be committed in a sequential run of the same simulation, and every message or event in a sequential run of the simulation must be matched by a message or event in the committed trace of the parallel run. Committing a message allows its buffer to be freed. Committing an event allows the associated state to be discarded.

TWOS simulations sometimes need to perform output to devices not directly under the control of TWOS. Since TWOS currently does not have its own file system, disk drives are an example of such devices. Data written to devices not under TWOS control cannot be rolled back. Therefore, when a user requests such a write, TWOS must delay the actual I/O until the write request is certain to be correct. That certainty is obtained when the event performing the write is committed. Therefore, write requests are tagged with the virtual time of the event requesting them and are held until their commit point is reached. Output in TWOS uses the same message mechanism as event scheduling, so cancellation of rolled back output is straightforward. In general, TWOS must delay performing any action it cannot undo until the commit point for that action is reached.

TWOS periodically runs a calculation to determine which messages and events can be considered committed. Essentially, anything earlier than the earliest unprocessed event will never be rolled back. The virtual time of that earliest unprocessed event is called global virtual time, or GVT. TWOS calculates a conservative estimate of GVT so that it can free storage used by committed messages and events that need no longer be saved.

TWOS runs on several different architectures, including the BBN Butterfly GP1000 parallel processor (which uses 68020 processors) and networks of Sun workstations. Typically, TWOS runs on top of an existing operating system, but uses that system solely for its message passing facilities and as a system development environment.

TWOS includes a number of features not required for a very basic distributed discrete event simulation system. These include dynamic creation and destruction of objects, dynamic memory allocation, and dynamic load management. TWOS also contains many switches permitting optimistic execution to proceed in slightly varying ways, allowing experimentation with a number of different internal mechanisms.

3. Experiences With the Development of TWOS

Using optimistic execution as the sole synchronization mechanism in TWOS has had far reaching consequences. Most aspects of the system were affected in some way by this choice. Designers planning to incorporate optimistic synchronization into their systems should be aware of the far-reaching consequences of that choice.

The decisions necessary to produce a good distributed system using the paradigm of optimistic execution can be broadly divided into those regarding correctness and those regarding performance. Some decisions affect both. Correctness is of primary importance, but the possible advantage of optimistic execution over more conservative synchronization lies in performance, so choices regarding performance are of almost equal importance.

While TWOS is specifically designed as a discrete event simulation engine, the lessons learned about the use of optimistic synchronization can be applied more generally. Where a TWOS simulation is decomposed into objects, more general applications might be decomposed into processes. The state of a TWOS object is simply the data area of a process, and the message communications between TWOS objects are similar to explicit messages between processes. The TWOS terms will be used consistently throughout the paper, to avoid confusion, but the experience discussed applies to the more general case.

Other implementations of Jefferson's virtual time mechanism for distributed discrete event simulation exist. They include a shared memory implementation by Fujimoto [Fujimoto 89], Jade's commercial implementation of Time Warp [Lomow 88], a LISP version written by Rand [Burdorf 90], and a C++ version that runs on the Caltech/JPL Mark 3 Hypercube [Steinman 91]. These implementations all differ somewhat from TWOS in various ways. TWOS is the implementation containing the most experimental features of any of them, and TWOS' performance has been more thoroughly studied than any of the other implementations.

3.1 Ensuring Correctness in Optimistic Synchronization Systems

Correctness problems with TWOS arose from several sources. These included:

- Improper rollback
- Failure to make progress in the computation
- Non-deterministic execution
- Improper commitment

Each of these areas will be examined separately.

3.1.1 Improper Rollback

The primary source of synchronization-related correctness errors in the development of TWOS was improper rollback. When the system detects a synchronization error, the work done out of order must be rolled back and the affected part of the system restored to the state it was in before the synchronization error occurred. Conceptually, this amount to restoring the state of the object that ran an event out of order and cancelling any side effects that its event may have caused.

Restoring the state is conceptually simple. By keeping multiple copies of the object's state, the system need only find the proper copy of the state and replace the current copy with that proper copy. If full copies of states are kept, the process is very simple. If only changes in the state are kept, then the restoration of a proper state is slightly more complex, but still fairly straightforward, in principle. Changes are undone, starting from the most recent, until the proper state is restored.

Side effects are much trickier to undo, especially if those side effects can cause further events to be run at other objects. Tracking down every possible side effect of a computation may sound very complicated, and it can be, unless care is used in defining what side effects a computation can have. TWOS carefully ensures that the only possible side effect from a computation is the sending of one or more messages. Any effect that the user wished to produce, be it scheduling another event or performing output to an I/O device, is cast in the form of a message. By limiting the problem to a single class of side effects, undoing them proved much easier.

TWOS undoes its only permitted form of side effect by message cancellation. TWOS keeps a copy of any message sent by an object. If the event that sent a message is rolled back, the copy of the message serves as a record that the message was sent, and signals the necessity of cancelling that message. For correctness purposes, the entire message need not be saved, just sufficient information to permit its cancellation. (Such information would include its destination and some form of unique identifier.) TWOS keeps the full messages for performance purposes, as will be discussed in section 3.2.3.

If an optimistic execution system permits any other class of side effects, then a separate mechanism for undoing them must be supported. The TWOS experience suggests that using a single side effect mechanism is much simpler. Any side effect that the user needs to perform can be done by sending a message to a special object capable of performing that side effect. If message cancellation, rollback, and commitment are properly implemented, cancellation of the side effect then becomes simple.

One natural consequence of this limitation is that system designers cannot rely on existing libraries or utility programs to provide support to users. As an example, dynamic memory allocation in TWOS could not be provided using the UNIX `malloc()` call, even in those cases where TWOS ran on top of a UNIX operating system. A `malloc()` call cannot be rolled back, as it is outside the scope of the optimistic execution system. It could be effectively undone by deallocating the space requested, but then the optimistic execution system would need to keep track of all `malloc()`'s that were made. (`malloc()` has an additional problem for an optimistic execution system, in that it returns a pointer to actual memory locations. Dynamically allocated memory is essentially an expansion of a process' state, so the optimistic system will need to keep multiple versions of dynamically allocated memory segments. Using `malloc()` to get these segments complicates keeping track of multiple versions and ensuring that the correct version is available upon rollback.)

Generally, then, existing utilities cannot be used by optimistic execution systems unless the systems designers are quite sure that those utilities lead to no side effects, or the designers are prepared to keep track of calls to the utilities and undo any effects that they may have. One other option is to encapsulate the utility in a separate object and invoke it via a message. The actual invocation of the utility is delayed until the message is committed, thereby assuring that the request to run the utility will not be rolled back. This option will work well, provided the utility is not expected to return information to the caller. If the utility does return information, it will not do so until the commit point has been reached, delaying proper execution at the caller until that time. The caller may perform other events, but they will be rolled back once the utility returns its information. Utilities returning values that are to be used immediately should be written by the system designers, so that they can be subject to rollback and need not wait for commitment.

3.1.2 Failure To Make Progress

Optimistic synchronization systems can be proved to always make progress towards completion, provided certain simple criteria are met [Jefferson 82]. These include no messages sent into the past, no infinite loops of messages whose virtual receive time equals their virtual send time, a guarantee that all messages will eventually be delivered, and a scheduling mechanism that ensures that the event in the system with the current lowest virtual time will eventually be scheduled. Trapping messages sent into the past and messages whose virtual send and receive times are the same is relatively easy. Many known message passing mechanisms that guarantee delivery can be used. The scheduling requirements deserve some further consideration.

The most typical scheduling policy used in virtual time mechanisms is to force each node to schedule the local event with the lowest timestamp first. If preemption of running events is used, this scheduling policy will make the necessary guarantee. Other policies could also be used, including time slicing policies and policies based on the probability of the an event being correctly executed, as estimated by some heuristic. The major differences between the many correct scheduling policies is their effect on system performance. Preemptive lowest virtual time first has proven to perform well in most cases [Burdorf 90]. Whatever scheduling policy is chosen, however, the system designers must be certain that it will never indefinitely delay the lowest timestamped event in the system.

3.1.3 Non-Determinism

In many, though not necessarily all, distributed systems, determinism is desirable at the user level. If a user runs a program twice in a row with the same inputs, he has every right to expect to get the same outputs. Optimistic execution systems can achieve determinism, with some care [Reiher 90a], [Mehl 92]. Areas to be careful of to assure determinism are ordering of messages and events, and initialization of data areas. Determinism in an optimistic synchronization system depends on always presenting a set of inputs in exactly the same condition, with respect to order and contents. Lack of caution in ordering of messages can ruin determinism. Also, all buffers and data structures should be properly cleared before being presented to the user. Otherwise, uninitialized garbage can cause an event to produce different results on two different runs. Experience shows that users cannot avoid these sorts of problems with reasonable care, so the system must make certain that the problems do not arise.

Many existing non-optimistic distributed systems do not guarantee determinism, so some optimistic distributed systems might not have to, either. Inserting non-determinism into an optimistic system is really quite easy – if you aren't very careful, it will be non-deterministic. However, mere carelessness is not the proper approach, even if determinism is not required.

Such systems can safely relax some issues concerning ordering of messages. At the minimum, they could permit messages with the same timestamps to be queued in varying orders, depending on the order of their arrival. If more non-determinism is still permissible, some method of permitting out-of-order arrival of messages to avoid rollback, provided they are not too out-of-order, could allow more performance gains. Some thought would be required here to determine how far out-of-order such arrivals could be before a rollback would be required. Generating virtual timestamps from loosely synchronized local clocks can also result in non-deterministic operations that do not have too many bad characteristics.

Even if full determinism is not required, the system designers should be careful about initializing buffers and data structures before giving them to users. The non-determinism that can result tends to be much more unpredictable and harmful than that caused by slightly varying orders of message queueing.

3.1.4 Commitment Errors

Some care must be taken with the commitment protocol of an optimistic synchronization system. Any data item that is determined to be committed can be deleted, so if the commitment protocol mistakenly sets its GVT too high, data required for a rollback might be deleted. Also, only those irreversible actions whose timestamp is earlier than GVT can be performed. If GVT is calculated too high, an irreversible action that should be rolled back might be performed, leading to errors. On the other hand, if the calculation of GVT is too conservative, commitment will be delayed far longer than necessary and system performance can suffer.

The primary difficulty in calculating GVT is ensuring that a consistent snapshot of the GVT contributions of every node is made. Because it is a distributed computation, a carelessly written GVT algorithm can fail to consider a message sent in the middle of GVT calculation, leading to an incorrectly high estimate of GVT. [Bellenot 90] discusses some of the problems of correctly estimating GVT and presents an efficient algorithm for doing so.

3.2 Performance Issues for Optimistic Synchronization Systems

Optimistic synchronization systems can achieve very good performance. On certain irregular simulations, TWOS has achieved speedups in excess of 30 (on 72 nodes), which is nearly 60% of the theoretically available speedup, as determined by critical path analysis [Presley 89]. Other discrete event simulation mechanisms have not been able to achieve comparable speedup on similar simulations. However, performance of this caliber is highly dependent on design choices. Poor choices can diminish performance, or limit the domain of applications in which good performance is achieved.

A major factor in obtaining good performance is overhead, as it is for any system. Usual methods for minimizing overhead should also be applied to the design and tuning of an optimistic synchronization system. However, optimistic synchronization systems have overheads not present in other distributed systems, and these overheads must be minimized, as well.

In addition to overhead considerations, other design choices in scheduling, message handling, and memory management can have impact on the performance of the system. Making good choices or poor choices here can mean the difference between good performance and mediocre performance.

Finally, the optimistic execution paradigm has certain optimization possibilities. These rarely allow major gains in performance, but they can be worthwhile in some cases.

3.2.1 Overhead in Optimistic Synchronization Systems

There are several components of overhead in optimistic execution systems. They include

- Message sending overhead
- Rollback overhead
- State saving overhead
- GVT calculation overhead

The following sections discuss these overhead components.

3.2.1.1 Message Sending Overhead

Message sending overhead is a major component of the total overhead of TWOS. Objects communicate only by sending messages, and events are scheduled only by receiving messages. Like most message-based systems, decreasing the cost of sending and receiving a message is likely to benefit optimistic synchronization systems. If the communications subsystem is asynchronous, as it is for TWOS, latency is of somewhat less concern than processing time. The sender does not block, and more often than not the receiving processor will have some other event to run while waiting for the delivery of this message. At the extreme, of course, long latency will be harmful, as it will tend to cause objects to run far into their futures before they receive a message rolling them back. In TWOS, on the BBN GP1000 parallel processor, the minimal overhead for sending a zero-byte message is .97 milliseconds, and the minimal latency for delivery of an off-node message is 2.3 milliseconds, measured from the point at which the message is presented to TWOS' source node to the time the destination node puts it in the receiving object's input queue. These overhead figures are sufficiently low to allow good performance on the GP-1000, which has Motorola 68020 nodes. Faster nodes may require lower overheads to provide good performance on the same types of applications.

3.2.1.2 Rollback Overhead

A common reaction to the idea of optimistic execution is that the cost of the rollbacks will be prohibitive. In actual fact, rollbacks themselves are fairly cheap. Rolling back a TWOS object requires searching an ordered state list for the proper entry (usually a short list, and usually starting from a point close to the proper entry); switching a pointer to the proper state; changing two or three variables in a control structure; putting the rolled back object into the (usually short) scheduler queue in the (usually close) proper place; and sending out any cancellation messages. The cost of a very minimal rollback on TWOS is .2 milliseconds, considerably less than the cost of sending a message.

The only part of the rollback that frequently has significant expense is the sending of cancellation messages. In TWOS, the cancellation messages are already completely set up for delivery in the rolled back object's output queue, so the costs are in identifying them and sending them. Identifying them usually requires a search of a short list, starting close to the point being searched for. The overhead of sending the cancellations, however, is almost the same as sending the normal message.

The dominating overhead cost of a rollback, then, is in the sending of any cancellation messages. Minimizing the cost of sending messages will help here, as well as in normal operations. Also, an optimization called lazy cancellation, discussed in section 3.2.3, can cut down on the costs of cancellation.

3.2.1.3 State Saving Overhead

A more significant overhead cost is state saving. TWOS saves multiple states of each object so that the objects can be rolled back, should they receive a message for an earlier virtual time. Saving a state requires allocating a chunk of memory to hold the saved version of the state, copying the bytes of the state into that chunk of memory, and inserting the saved state into a queue of saved states. Both the memory allocation and the copying costs can be high, depending on circumstances. The minimal cost of saving a zero byte state in TWOS on the GP1000 is .26 milliseconds, plus .23 milliseconds per 1000 bytes of state. The cost can rise significantly, however, if the system has trouble finding a sufficiently large chunk of memory to hold the state.

One method of cutting the cost of state saving is to only save changes in the state. If very little of the state is changed in a typical event, a much smaller chunk of memory is required to hold the changes, and the cost of copying the changes will be much lower than copying the whole state. Because TWOS does not run on bare hardware, it does not have access to the page tables, and cannot examine dirty bits or trap writes. Thus, TWOS is not able to detect which parts of a state have changed during an event, and must save the entire state. If an optimistic execution system is being built on top of bare hardware, using page table information to save only changed portions of a state could greatly lower the costs of state savings. [Fujimoto 88] proposed special hardware support to handle this problem, in the form of a *rollback chip* that would automatically detect changes to portions of a state and save earlier versions in a way that would make rollback cheap and easy. This rollback chip, if well integrated into the optimistic synchronization system, could largely eliminate the overhead of state saving.

Another way to reduce the costs of state saving would be to require users to identify which portions of the state change during an event. Only those portions would need to be saved. Traditional programming languages would require much effort on the part of users to correctly identify which portions of their state changed, and failing to inform the system that part of the state changed could lead to very tricky errors. However, object-oriented programming languages like C++ offer facilities for making the identification of changed state much simpler. [Steinman 91] has had good success with using C++ with his optimistic execution system to cut the costs of state saving in this way.

Yet another method of minimizing state saving overhead is to not save an object's state after every event. Instead, it can be saved every other event, or every third event. If the object doesn't roll back, or rolls back to one of the saved states, this method works well. If the object rolls back to one of the events whose state was not saved, the rollback must go further, back to the next earliest event whose state was saved. The system then re-executes events that did not actually need to be redone until it regenerates the state needed by the event that was actually rolled back.

Both analytic work ([Lin 89]) and experimental results ([Bellenot 92], [Preiss 92]) suggest that periodic state saving can improve performance in certain cases, but can degrade it in others. If the chances of rollback are low and the cost of saving states high, periodic state saving wins. If the chances of rollback are high and the cost of saving states low, periodic state saving loses. In particular, if every processor hosts large numbers of fairly active objects, periodic state saving will probably do well, as the processor's time is better spent running an event than preparing for a rollback that probably won't come. On the other hand, if each processor has only a few objects, the chances that the next event a processor wants to execute is properly synchronized are much lower, and the processor would be better off spending its time preparing to lessen the cost of a fairly likely rollback.

Some have suggested, in the latter case, that the processor is better off not running an event that is very likely to be rolled back. However, if the choice is between running an event that is likely to be rolled back and doing nothing, experience with TWOS has shown that running the event is usually better [Reiher 89]. Running an event with a small chance of being correct is generally a more productive use of the processor's cycles than doing nothing at all. Of course, the processor is also creating overhead for other processors that may have better work to do, by sending them messages that are fairly likely to be cancelled. However, except in low-memory situations, those costs seem to be offset by the benefit of occasionally running a correct event. This issue will be discussed in more detail in section 3.3.2.3.

3.2.1.4 GVT Calculation Overhead

The primary purpose of calculating GVT is to permit the system to fossil collect old states and messages no longer needed to support possible rollbacks. The more frequently GVT is calculated, the quicker such items' memory is returned to the heap for use by other events. On the other hand, the GVT algorithm requires multiple phases of message sends, with some calculations on each node to determine local contributions to GVT. With GVT calculations performed every second, the existing TWOS GVT algorithm spends considerably less than 1% of the total execution time of the system running GVT-related code. An earlier, less clever algorithm, which used substantially more messages and longer delays in calculation, had around a 1% impact on the total elapsed time of typical simulations, suggesting that even a bad GVT algorithm will not have terrible effects on the performance of the system [Bellenot 90].

3.2.2 Performance Design Choices For Optimistic Synchronization Systems

Some of the design choices for an optimistic synchronization system that can have profound effects on performance include:

- Scheduling and Priority Mechanisms
- Memory Management Strategies
- Load Management and Migration

The following sections discuss these issues in more detail.

3.2.2.1 Scheduling and Priority Mechanisms

The virtual time tags used in optimistic execution systems serve as rough indicators of the priority of events and messages, and as equally rough indicators of the probability of events and messages being correct. An important heuristic for getting good performance out of an optimistic execution system is to favor operations with low virtual time tags over operations with high virtual time tags.

As discussed in section 3.1, many scheduler policies can give correct results for optimistic execution systems. However, the policy that has given the best results for TWOS, and most other optimistic execution system, is preemptive lowest virtual time event first. [Burdorf 90] investigated some particular cases for various scheduling policies under the Rand implementation of Time Warp, and found that no tested policy did very much better than lowest virtual time first. There is a superior policy, however. Ideally, the next event scheduled by each node should be the event least likely to be rolled back. As yet, lowest virtual time first is the best heuristic known for estimating the probability of correct execution. Research on better heuristics is ongoing [Steinman 92].

Giving preference to low virtual time items is used heuristically throughout TWOS to increase performance. Generally, the lower the virtual time tag on an item, the more likely it is to be correct and the more likely it is that delaying it will cause incorrect execution that will have to be rolled back. Therefore, message routing, memory management, and other system functions are all driven by timestamps. A message with a low timestamp will be delivered before a message with a high timestamp. In tight memory situations, a request with a low timestamp will be given preference over a request with a high timestamp.

Priority should also be given to negative message traffic. Negative messages are sent when an event is rolled back and TWOS needs to cancel the messages it sent. Such negative messages are certain indicators that their corresponding positive messages are incorrect, so any events performed by those positive messages are certain to be rolled back. Therefore, getting the negative messages to their destinations to cancel the positive messages as quickly as possible will greatly reduce the number of rollbacks. TWOS always gives delivery priority to any negative message over any normal message.

3.2.2.2 Memory Management Strategies

Optimistic execution systems, in one view, trade space for time. By keeping multiple copies of data items, they can run applications faster. The cost is high memory utilization. Optimistic execution systems are likely to place a much higher strain on memory management than other systems.

Because TWOS does not have access to page tables and other low level hardware, TWOS does not itself run a virtual memory system. Some of the systems TWOS runs on top of do support virtual memory, but they remove control from TWOS entirely whenever they detect a page fault. In most cases, TWOS could do some other useful work while waiting for its page, so frequent page faults lead to poor TWOS performance. Therefore, TWOS tries to limit itself to the physical memory actually available.

Since optimistic execution tends to use up memory, TWOS must be prepared to deal with situations in which a processor has filled its available memory with data with high timestamps, but then must satisfy a memory request with a low timestamp. The full theory on how to correctly handle this situation is outlined in [Jefferson 91a], but, in brief, the system should discard data items with high timestamps to make space for items with low timestamps. TWOS has this protocol, called *cancelback*, partially implemented, and it permits the system to complete applications that would otherwise have failed from memory exhaustion. Any optimistic execution system that is constrained to limited memory should have this protocol, or some variant, in place.

If the optimistic execution system has sufficient control of the hardware to handle virtual memory itself, many of the problems of limited memory can be alleviated. In such cases, the virtual memory system should take virtual timestamps into account when performing page replacement. Generally speaking, pages with low timestamps are more important than pages with high timestamps, but the designers must also take into account whether a given page is associated with an event that has already been performed or not. No optimistic execution system has yet had its own virtual memory system, so much remains to be learned in this area.

3.2.2.3 Load Management and Migration

Like many other distributed systems, optimistic synchronization systems can sometimes benefit from dynamic load management. TWOS uses dynamic load management to deal with irregularities in its applications [Reiher 90b]. There are several characteristics of optimistic synchronization that can complicate dynamic load management. First, many dynamic load management policies shift processes from machine to machine on the basis of the utilizations of the various machines. Simple utilization is not a good choice of policy parameter on an optimistic synchronization system, since such a system will try very hard to remain busy at all times, even though much of the work it does might be rolled back. Thus, few processors in optimistic synchronization systems will ever have low utilization.

One policy parameter that can take the place of simple utilization is *effective utilization*. Effective utilization is fully discussed in [Reiher 90b], but, briefly, it is an estimate of the proportion of time each processor spends doing work that is eventually committed. Only such work is of benefit to the application, so processors with low effective utilization are relatively underloaded, even if their simple utilization is quite high. Load balancing on the basis of effective utilization has proven very effective in TWOS.

Another problem likely to arise in dynamic load management for optimistic synchronization systems is that process migration may be quite expensive. Each process has multiple copies of its state and many messages associated with it. Moving all of this data from one node to another may take a long time. One method of dealing with this problem is temporal decomposition. Temporal decomposition permits the system to divide processes into subprocesses. Each subprocess takes responsibility for the process' activities for some span of virtual times. By splitting processes in this way, only the portion of the data related to one subprocess needs to be moved to another node. If the split is chosen so that most upcoming work for the process is in the timespan of the subprocess to be moved, the load effect of the migration will be nearly the same as moving the entire process. Splitting processes this way can complicate rollback, so that rolling back a single object can require sending a state from one of its subprocesses to another. Despite the costs and complications, temporal decomposition has proven valuable in limiting the cost of process migration in TWOS [Reiher 90b].

3.2.3 Performance Optimizations For Optimistic Synchronization Systems

Optimistic execution systems have several possible optimizations not available to other systems. These are based on the observation that work done out of order can sometimes produce correct results, anyway. An event run out of order might send correct messages, and might produce a correct state for the next event. When an event is rolled back, instead of discarding its state and cancelling its messages, they can be saved until the event is executed again. If the re-execution of the event sends the same messages, there is no need to send them again or cancel them. If different messages are sent, the new ones are sent out and the old ones that were not resent are cancelled. Similarly, if the new version of the state is the same as the old version of the state, any subsequent events performed by this object before the rollback worked with a correct input state, and thus need not be redone.

When applied to messages, this optimization is called *lazy cancellation*. It has been given several names when applied to states. The named used in TWOS is the *jump-forward optimization*. Both of these optimizations have been implemented in TWOS.

Another class of optimizations is based on the belief that sometimes optimistic systems can be too optimistic. [Lubachevsky 91] describes several circumstances in which fully optimistic execution can cause severe performance problems. In some cases, throttling optimistic execution may prove beneficial.

3.2.3.1 Lazy Cancellation

Lazy cancellation proved to be worthwhile for most applications, in the sense that it avoided unnecessary message cancellations often enough to make the application run faster [Reiher 90c]. In order to improve performance, lazy cancellation must save cancellations fairly often, as it requires comparisons of messages and extra scanning of queues. The minimal overhead to check for lazy cancellation when it will not occur is .25 milliseconds in TWOS, and it would be more expensive for realistic cases, so it needs to win fairly frequently. In typical TWOS runs, lazy cancellation can save the resending of tens of thousands of messages.

On the other hand, lazy cancellation can perform poorly for certain applications. If running an event out of order cannot possibly result in a correct message, then lazy cancellation will never save any message resending and any overhead cost it incurs will simply delay the simulation. In addition, delaying the cancellation of messages may permit the objects that received them to improperly run other events, possibly in the place of correct events. Because lazy cancellation can sometimes perform very poorly, TWOS also supports the alternative, *aggressive cancellation*, as a run time option. Since existing applications make good use of lazy cancellation, that option is the default.

3.3.3.2 The Jump-Forward Optimization

The jump-forward optimization was put into TWOS on the assumption that it might provide a benefit similar to lazy cancellation. Unfortunately, it did not. Statistics indicated that it did, indeed, occasionally save the reprocessing of an event, but it did not do so very often. No performance penalty was seen through the use of the jump-forward optimization, but its presence substantially increased the complexity of TWOS code, so it was eventually removed. Obviously, for certain cases this optimization would work extremely well, so other systems developers working on optimistic synchronization might want to examine their applications to determine whether the jump-forward optimization is likely to win in their situations.

An interesting side note about both lazy cancellation and the jump forward optimization is that both can permit applications to run faster than critical path analysis suggests is possible. Critical path analysis works on the assumption that no event on the critical path can produce useful results until all earlier events on the critical path have been performed. These two optimizations can sometimes permit a critical path event to produce correct results before those other critical path events have completed, thereby speeding up the total critical path. Supercritical speedup has never been seen in a realistic application, but it has been reproduced in artificial applications [Jefferson 91b]. Developers of optimistic synchronization systems should not count on achieving supercritical speedup, but use of these optimizations can permit increased performance through local supercritical speedup of some parts of the application.

3.3.2.3 Throttling

Some optimizations for optimistic execution are based on throttling the optimism of the system, on the theory that performance suffers by executing too far into the future. In the zero-overhead, unlimited resource case, the value of throttling optimism is low, but it can sometimes prove valuable in more realistic cases. Throttling has proven of value in two cases in TWOS, so far. The first is when certain objects use up all of a processor's memory while executing very far beyond the times other processors are handling. Eventually, memory management has to be invoked on the filled processor, leading to substantial performance penalties. The second case in which throttling has proven valuable is when overly optimistic execution causes an object to send large numbers of messages far into the future, causing its output queue to grow very large and thereby increasing queueing time. (This second case might be better solved with an improved data structure for the queue, rather than throttling optimism.)

The proper method of throttling optimistic synchronization systems remains a topic of research. One method suggested by [Sokol 90] is to set up a time window, so that no processor can execute events more than a certain distance into the future. Existing implementations of this method have given some performance improvement in certain cases, but other tests have not shown any performance improvement with this method [Reiher 89]. Time window mechanisms typically require user knowledge of the proper window size, and can unfairly penalize processes that are doing useful work. Methods that do not require user intervention are greatly preferable.

One such method is described in [Madisetti 92]. In this optimistic execution system, periodically, on a probabilistic basis, some portion of the uncommitted work in the system is discarded, thereby guaranteeing that any overly optimistic work will be pruned. This system has provided good performance results for certain applications in a shared memory environment, and does not necessarily require that the user provide any simulation-specific knowledge to the throttling mechanism. Further research is necessary to determine if the method works well for broader classes of simulations, and whether it can be reasonably implemented on distributed memory machines.

4. Conclusions

Optimistic synchronization systems offer performance advantages for many important distributed systems applications. However, to achieve their complete benefit the optimistic synchronization mechanism must be in control of most of the activities of the system. Since optimistic synchronization systems work very differently than more common distributed systems, correctly and efficiently implementing a complete optimistic synchronization system is a challenging task. As research in this area progresses, the task will become more familiar and better understood.

The major correctness issues in the design of an optimistic synchronization system are control of side effects for rollback purposes, ensuring that the system makes progress through proper scheduling and handling of messages, maintaining determinism (if it is required), and proper commitment. These issues are well understood, with the possible exception of control of side effects. As long as side effects are limited to messages, and any more complex side effects are delayed until they are committed, the Time Warp mechanism designed by Jefferson will handle them correctly. If more complex side effects before commitment are required, more care will have to be taken.

The performance issues in designing optimistic synchronization systems are not as well understood, though much work has been done on them and research continues. Under the proper conditions, an optimistic synchronization system can achieve very good performance even with highly irregular applications. But there are still circumstances that can lead to poor performance despite the potential for good performance. Methods that can lead to better performance include reduction of the most important sources of overhead; scheduling issues; memory management; and dynamic load management. In the area of overhead reduction, the most profitable areas to examine are message sending costs and state saving overheads. Various optimizations to the basic optimistic synchronization protocol can provide secondary gains.

In some cases, distributed systems designers may be able to include an optimistic component in an otherwise standard system. If the component has the proper characteristics, using optimistic synchronization to handle it may improve its performance significantly. Generally, if an operation can often, but not always, be performed correctly with the local data currently available, it may be a promising candidate for optimistic synchronization. The other important requirements are the ability to recover correctly and efficiently if the optimism proves unfounded, and ease in assigning meaningful timestamps to all relevant operations to allow detection of incorrect synchronization. [Goldberg 92] describes how certain data replication problems fit these criteria, and how optimistic synchronization can be used to handle such problems.

Optimistic execution methods can be applied to systems outside the field of discrete event simulation. Distributed databases can use full optimistic synchronization to control their operations, as discussed in [Jefferson 84], with possible advantages in performance and ease of implementation of difficult features, like replication and nested transactions. Programming language support for optimistic synchronization may permit it to be used for general purpose programs run in a parallel or distributed environment. A number of efforts are underway to provide such support. Whether or not optimistic synchronization is an appropriate way to provide all synchronization support for a full-scale, general purpose distributed operating system is an open question.

Availability

The Time Warp Operating System is available through the NASA Cosmic Software Distribution system. The address of Cosmic is

Cosmic
The University of Georgia
382 East Broad Street
Athens, GA 30602

The basic version TW 2.0 is available, as is version 2.5, which includes dynamic load management. An educational discount is given to universities.

Acknowledgements

This work was funded by the U.S. Army Model Improvement Program (AMIP) Management Office (AMMO), NASA contract NAS7-918, Task Order RE-182, Amendment No. 239, ATZL-CAN-DO.

Bibliography

[Bellenot 90] Steven Bellenot, "Global Virtual Time Algorithms," *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Volume 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Bellenot 92] Steven Bellenot, "State Skipping Performance With the Time Warp Operating System,"*Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Bhargava 82] Bharat Bhargava, "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and Its Comparison To Locking," *Proceedings of the IEEE Conference on Distributed Computer Systems*, 1982.

[Burdorf 90] Christopher Burdorf and Jed Marti, "Non-Preemptive Time Warp Scheduling Algorithms," *Operating Systems Review*, Volume 24, No. 2, Apr. 1990.

[Carey 88] Michael J. Carey and Miron Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proceedings of the 14th Conference on Very Large Databases*, 1988.

[Chandy 79] K. Mani Chandy and Jayadev Misra, "Distributed Simulation: A Case Study In Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, SE-5 (5), Sep. 1979.

[Fujimoto 88] Richard Fujimoto, et al, "Design and Performance of Special-Purpose Hardware For Time Warp," *Proceedings of the 15th Annual International Symposium of Computer Architecture,*, 1988.

[Fujimoto 89] Richard M. Fujimoto, "Time Warp on a Shared Memory Multiprocessor", *Proceedings of the 1989 International Conference on Parallel Processing*, Aug.1989.

[Fujimoto 90] Richard Fujimoto "Parallel Discrete Event Simulation," *Communications of the ACM*, Vol. 33, No. 10, Oct. 1990.

[Goldberg 92] Arthur Goldberg, "Virtual Time Synchronization of Replicated Processes" *Proceedings of the 1992 SCS Conference on Distributed Simulation,* Volume 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Jefferson 82] David Jefferson and Henry Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control," Rand Note N-1906AF, The Rand Corp., Santa Monica, Cal., Dec. 1982.

[Jefferson 84] David Jefferson and Ami Motro, "The Time Warp Mechanism for Database Concurrency Control," Technical Report TR-84-302, Department of Computer Science, University of Southern California, Jan. 1984.

[Jefferson 87] David Jefferson, Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger and Steve Bellenot, "Distributed Simulation and the Time Warp Operating System", *11th Symposium on Operating Systems Principles (SOSP),* Nov. 1987.

[Jefferson 91a] David Jefferson "Virtual Time II: The Cancelback Protocol," *Proceedings of the Conference on Principles of Distributed Computing*, 1990.

[Jefferson 91b] David Jefferson and Peter Reiher, "Supercritical Speedup," *Proceedings of the 24th Annual Simulation Symposium*, Apr. 1991.

[Lin 89] Yi-Bing Lin and Edward Lazowska, "The Optimal Checkpoint Interval In Time Warp Parallel Simulation," Technical Report 89-09-04, Department of Computer Science and Engineering, University of Washington, Sep. 1989.

[Lomow 88] Gregory Lomow, John Cleary, Brian Unger, and Darrin West, "A Performance Study of Time Warp," *Proceedings of the 1988 SCS Conference on Distributed Simulation,* Volume 19, No. 3, Society for Computer Simulation, Jan. 1988.

[Madisetti 92] Vijay K. Madisetti, David A. Hardaker, Richard M. Fujimoto, "The MIMDIX Operating System for Parallel Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Mehl 92] Horst Mehl, "Breaking Ties Deterministically In Distributed Simulation Schemes," Technical Report 217/91, Department of Computer Science, University of Kaiserslautern, Federal Republic of Germany, Dec. 1991.

[Preiss 92] "On the Trade-off Between Time and Space in Optimistic Parallel Discrete Event Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Presley 89] Matt Presley, Maria Ebling, Fred Wieland, and David Jefferson, "Benchmarking the Time Warp operating system with a computer network simulation", *Proceedings of the 1989 SCS Conference on Distributed Simulation*, Volume 21, No. 2, Society for Computer Simulation, Jan. 1989.

[Reiher 89] Peter Reiher, Frederick Wieland, and David Jefferson, "Limitation of Optimism in the Time Warp Operating System", *Winter Simulation Conference*, Washington, D.C., Dec. 1989.

[Reiher 90a] Peter Reiher, Frederick Wieland, and Philip Hontalas, "Providing Determinism in the Time Warp Operating System – Costs, Benefits, and Implications," *Proceedings of the Second IEEE Workshop on Experimental Distributed Systems*, , Oct. 1990.

[Reiher 90b] Peter Reiher and David Jefferson, "Dynamic Load Management In the Time Warp Operating System," *Transactions of the Society For Computer Simulation*, Vol. 7 No. 2, Jun. 1990.

[Reiher 90c] Peter Reiher, Richard Fujimoto, Steven Bellenot, and David Jefferson, "Cancellation Strategies in Optimistic Execution Systems", *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Vol. 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Sokol 90] Lisa Sokol and Brian Stucky, "MTW: Experimental Results For a Constrained Optimistic Scheduling Paradigm," *Proceedings of the 1990 SCS Conference on Distributed Simulation*, Vol. 22, No. 2, Society for Computer Simulation, Jan. 1990.

[Steinman 91] Jeff S. Steinman, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation," *Proceedings of the 1991 SCS Conference on Parallel and Distributed Simulation*, Vol. 23, No. 1, Jan. 1991.

[Steinman 92] Jeff S. Steinman, "SPEEDES: A Unified Approach To Parallel Simulation," *Proceedings of the 1992 SCS Conference on Distributed Simulation*, Vol. 24, No. 3, Society for Computer Simulation, Jan. 1992.

[Strom 90] Robert Strom, "Hermes: An Integrated Language and System for Distributed Programming," *Proceedings of the Second IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, Oct. 1990.

Experiences from Multithreading System V Release 4

J. Kent Peacock
Sunil Saxena
Dean Thomas
Fred Yang
Wilfred Yu

Intel Multiprocessor Consortium
2801 Northwestern Pkwy.
Santa Clara, CA 95052-8122
Email: {kentp,sunil,fred,wyu}@stps18.intel.com
deant@ctnews.convergent.com

ABSTRACT

An Intel-sponsored Consortium of computer companies has developed a multiprocessor version of System V Release 4 (SVR4MP), which has been formally released by UNIX System Laboratories. The Consortium's goal was to add fine-grained locking to SVR4 with minimal change to the kernel, and with complete backward compatibility for user programs.

Although the literature already abounds with largely similar descriptions of multiprocessor UNIX kernels, multithreading SVR4 presented additional challenges which required some new approaches. A sampling of the most important problems and their solutions are given. In particular, the locking model, debugging tools, interrupt-related deadlock avoidance, TLB shootdown and file system cache scalability are touched upon.

To test the scalability of the resulting kernel, the *GAEDE* benchmark, which is fairly I/O intensive, was used. The kernel was found to obtain throughput scalability of 88% using 5 processors.

1. Introduction

The goal of the Consortium assembled by Intel was to produce a multiprocessor version of System V Release 4 in as short a time as possible. As such, there was no inclination to perform a radical restructuring of the kernel, nor to support user-level threads while the evolution of a consensus threads standard was incomplete.

There were two main performance goals for this effort: binary performance running the *GAEDE* benchmark [6] with respect to the uniprocessor system should degrade by no more than 5%; and the proportional increase of throughput should be at least 85% of the first processor for each processor brought online, up to 6 processors.

Many attempts have been made to adapt UNIX to run on multiprocessor machines. Companies such as AT&T, Encore, Sequent, NCR, DEC, Silicon Graphics, Solbourne and Corollary have all offered multiprocessor UNIX systems, though not all have described the changes made to the operating system in the literature. Bach [1] describes a multiprocessor version of System V released by AT&T, with existing synchronization mechanisms replaced by Dijkstra semaphores. Encore has described several generations of their multithreading effort, first on MACH and then on OSF/1 in a series of papers [4, 10, 11]. DEC has also published papers on their approach to multithreading their BSD-derived ULTRIX system [9, 16]. Ruane's paper on Amdahl's UTS multiprocessing kernel is the best precedent to the philosophy of the approach used by the Consortium [13]. The paper gives a very good discussion of some subtle synchronization issues relative to a pre-SVR4 environment. Lately, NCR has also discussed their parallelization efforts on System V Release 4 [5]. As NCR has participated as a member of the

Intel Consortium, this work has influenced how the Consortium went about the multithreading task.

The remainder of this paper is divided into six sections. Section 2 describes the features of the locking primitives, which served as a foundation for the entire effort. Section 3 relates a particular difficulty due to the interaction of interrupt routines and locks. Section 4 explores the maintenance of coherent translation look-aside buffers among the processors in the system in the context of frequently changing system virtual address bindings. Section 6 describes an approach to reducing lock contention during access to a number of the caches maintained by the file systems, and is followed by some performance measurements indicating the scalability of the resulting system in Section 7. Finally, Section 8 gives a summary and some additional comments and observations which arose out of this experience.

2. Locking Model

As this effort was a commercial venture rather than a research project *per se*, one of the primary goals of this development was to minimize the amount of change made to the existing kernel code. To accommodate this goal, the locking strategy has been designed to fit with the existing uniprocessor synchronization mechanisms as much as possible. In particular, the semaphore model favored by some other UNIX multithreaders [1] was not considered to be a satisfactory choice. Good arguments against using Dijkstra semaphores to multithread UNIX are given in papers by Ruane [15] and Campbell *et al.* [5]. A locking strategy which complemented, rather than replaced, existing *sleep–wakeup* synchronization was found to be easier to fit into existing kernel code. The lock design evolved somewhat from the locks used by NCR in their first SVR4/MP effort [5]. Most of the functionality available in the Consortium locking primitives was present in the NCR locks. The main differences include a redesign of the actual interface functions, the inclusion of a processor priority argument on mutex locks (described below) and greater emphasis on the use of atomic operations.

2.1 Mutex Locks

The most important type of locking, mutex locking, provides a general mutual exclusion capability. There are some new features of the mutex primitives which provide a more general functionality than most previous multithreading efforts. Firstly, mutex locks are implemented so that any mutex locks held by a process are automatically released and reacquired across a context switch. This is an imitation of the implicit uniprocessor locking achieved by holding the processor in a non-preemptive kernel. It is necessary for proper *sleep–wakeup* operation for a lock protecting a sleep condition to be released after the sleeping process has established itself on the sleep queue. Previous efforts have enhanced the sleep function to release a single lock, usually specified as an extra argument to the sleep call [9, 15]. With releasing of locks across context switches, sleep calls need not be changed. Also, multiple locks can be held before the sleep call and all of them will be released atomically, as opposed to only a single lock with the additional sleep argument approach. This means that much of the multithreading effort merely involves adding mutex locks around sections of code, even though they may call sleep. In addition, it is possible to hold a lock across a call to a function which may in turn call sleep.

The ability to release all of the locks acquired in the call stack relates to the second important feature of the locks, namely that recursive locking of each individual lock is allowed. SVR4 is designed in such a way that there are a number of object-oriented interfaces between subsystems of the kernel. These subsystems call back and forth to one another and create surprisingly deep recursive call stacks. (This happens particularly between virtual memory and file system modules.) Though providing considerable modularity, this feature makes it very difficult to establish assertions about which mutex locks might be held coming into any given kernel function. Allowing lock recursion and the automatic release of mutex locks allows a given function to deal only with its own locking requirements, without having to worry about which locks its callers hold. This "locality of locking" allows effort to be applied to designing the locking and deadlock avoidance strategies, rather than tedious schemes to release and reacquire locks correctly. Without recursive locks available, a typical solution to the problem of recursive locking is to split functions apart into two versions: one which assumes that a given lock is held, and a second which acquires and releases the lock around a call to the first. Splitting functions apart is viable, although somewhat tedious, in a pre-SVR4 environment due to the less dynamic calling structure. This solution is not generally satisfactory for SVR4, however.

The lock extensions are also less necessary when the multithreading strategy is coarse-grained. A coarsely multithreaded kernel tends to be coded to release a coarse lock before calling into a different subsystem in order to reduce contention. However, the Consortium's fine-grained multithreading strategy calls for holding locks through such calls at times. This is done to avoid the overhead of unlocking and relocking and to preserve the integrity of the data structure protected by the lock. With sufficiently fine granularity, a given lock may be held longer without causing too much contention. However, the tradeoff with finer locking is the requirement for more subtle deadlock avoidance strategies.

The mutexes were also designed to be able to configure whether the caller spins or gives up the processor when the lock is not available. Locks acquired by interrupt routines must spin, since interrupt routines are not permitted to sleep. Requiring several locks which protect individual instances of the same data structure can potentially cause deadlock if the instances are not ordered. This type of deadlock can be avoided by sleeping to wait for a busy lock. The deadlock avoidance comes from the fact that the other held locks are released when a lock requester sleeps. This deadlock avoidance technique is used in multithreading the file systems, where some operations require holding mutex locks on several files at the same time. It is also possible to conditionally try to obtain a lock without waiting, in which case an error indiction is returned if the lock is not available.

The mutex locks possess additional properties besides these main characteristics. Most mutex locks in the system provide simple mutual exclusion. A mutex can also be configured as a shared/exclusive (or multiple reader/single writer) lock at initialization. Such a lock can be acquired either in shared or exclusive mode, with traditional semantics. It is also possible to pass a processor priority setting (*spl*) function to the mutex lock function. This allows raising processor priority around the holding of the lock. This is in recognition of the fact that some locks nest directly inside interrupt disabled code to provide a multiprocessor form of interrupt disabling. Passing the spl function to the locking function allows spin waiting for a busy lock to be done at the original priority, rather than the higher, passed-in priority. The locking function returns the old processor priority which, to preserve symmetry, can be passed to the unlocking function.

Figure 1 gives an example of some simple driver code fragments that illustrate these concepts. The processor priority setting functions can be directly replaced with mutex locks to provide mutual exclusion between interrupt routines and process-level code.

2.2 Additional Locking Functions

In addition to mutexes, which are used for the majority of locking, several other sets of locking primitives are provided. These primitives are characterized by the length of time that indivisibility is required.

The simplest primitives provide a set of atomic arithmetic and logical operations, all of which return the previous contents of the location operated on. The arithmetic functions are most useful for correct maintenance of reference counts on data structures. The return value is important in detecting the last decrement of the reference count in cases where the data structure should be freed or deactivated. The logical operations allow very fine multithreading (to the statement level) of bit manipulations of flag fields. Depending on the processor architecture, these primitives can often be implemented with a few machine instructions.

The next level of locking, where indivisibility is required for several statements, is provided by *simple locks*. Simple locks operate on a lock variable and are the simplest form of spin locking supported by the hardware. An example of their use would be to protect such operations as queue insertions and deletions. To avoid deadlock, these locks should not be held by a process when the CPU dispatcher is called, as they are not automatically released. These simple locks are practically identical to the *spinlocks* described by Ruane [15].

```
/* Original Uniprocessor Routines */
int devbusy;

devstart()
{
        register int oldpri;

        oldpri = spl5(); /* Exclude devintr interrupt */

        while (devbusy)           /* Wait for interrupt */
                sleep(&devbusy, PDEV);
        devbusy = 1;

        ....                      /* Start up device */

        splx(oldpri);             /* Restore old interrupt mask */
}

devintr()                         /* Interrupt routine at level 5 */
{
        ....                      /* Clean up device */
        devbusy = 0;
        wakeup(&devbusy);
}


/* Multithreaded Driver Routines */
int devbusy;
mutex_t devlock;

devstart()
{
        register int oldpri, spl5();

        /*
         * Exclude interrupt on this CPU and make it wait if it
         * occurs on another CPU. L_SPIN means spin-wait for the
         * lock. If the lock is busy, the spinning is done at
         * priority oldpri, rather than level 5.
         */
        oldpri = mutex_lock(&devlock, L_SPIN, spl5);

        while (devbusy)           /* Wait for interrupt */
                sleep(&devbusy, PDEV);
        devbusy = 1;

        ....                      /* Start up device */

        /* Unlock mutex and restore old interrupt priority */
        mutex_unlock(&devlock, oldpri);
}

devintr()                         /* Interrupt routine at level 5 */
{
        /* splnull means do not set processor priority */
        (void)mutex_lock(&devlock, L_SPIN, splnull);

        ...                       /* Clean up device */
        devbusy = 0;

        /* NULLSPL means no processor priority restore */
        mutex_unlock(&devlock, NULLSPL);
        wakeup(&devbusy);
}
```

Figure 1: Mutex Locks with *spl* Setting

The third locking level is that of the mutex locks, which were previously discussed. At the fourth and highest level are resource locks, which provide the capability to hold a lock for even longer time periods, particularly across context-switches. These locks are of interest in cases where more advanced locking may be necessary to support higher concurrency. With resource locking, the focus of the lock is a particular data structure which needs to be manipulated atomically for a long time. Resource locks can be used to replace traditional *sleep–wakeup* synchronization with more complex locking. For example, resource locking protocols which support multiple–reader and single–writer locking could be used to provide higher concurrency than simply placing mutex locks around the existing synchronization.

Although *sleep–wakeup* synchronization has many desirable properties, there is a serious performance problem which can arise, namely the "thundering herd" problem. This situation occurs due to the fact that *wakeup* makes all processes waiting on a given wakeup channel runnable, not just one. Ruane [15] gives a thorough discussion of this issue, with the interesting solution of putting all of the awakened processes onto a local run queue for the current CPU. This approach transiently serializes access to the resource being waited on through the local CPU's run queue. SVR4MP does not have a separate run queue for each CPU, although it could be argued that it should have for scalability reasons. The SVR4MP solution to this problem is to provide a primitive which allows the waking of only one process. The main drawback to this solution is that it requires code changes and places a burden of responsibility on the single awakened process to "pass it on" if it decides not to actually use the resource. On the other hand, the actual overhead of waking up and running only one process is likely to be lower when there are a large number of processes waiting on a sleep channel. This is true especially if the process which obtains the resource gives up the processor before releasing the resource. In Ruane's approach, the other waiters on the run queue would be dispatched uselessly to find the resource busy and go back to sleep. The thundering herd behavior would not be avoided.

2.3 Lock Debugging

Given the intention to perform fine-grained locking of the kernel, a number of tools were assembled to aid in the task:

- Multiprocessor kernel debugger with macro capability.
- General real-time trace event facility.
- Excessive lock spin (deadlock) detection.
- Automatic lock hierarchy generation and checking.
- Lock contention statistics.
- Lock call stack tracing.

All of these debugging aids have aided tremendously in identifying bugs, contention points and deadlocks. The automatic hierarchy checking facility generates the lock dependency graph dynamically while the kernel runs. This information has been useful in understanding the necessary ordering of locking and has allowed the addition of deadlock avoidance mechanisms where required (but not always anticipated). Clearly, this tool only provides a sub-graph of the complete lock dependency graph, which cannot be used to prove freedom from deadlock. When deadlocks do actually occur and cause a hung system, the excessive spin lock detection causes a trap into the debugger, where sufficient information is usually available to identify the cause.

The lock contention and call stack statistics have been used extensively to aid in tuning the kernel. Both sets of information are obtained by reading an appropriate lock debug device, which returns the information requested in ASCII format. Once a lock with high contention has been identified from the contention statistics, the call stack tracing can be used to provide a count of every call stack which locks a given lock. A nesting level of five function calls is provided. This allows the uses of a lock to be sub-divided and analyzed to determine the most productive changes to make.

One problem with the debugger is that it was not written as a stand-alone program, but uses kernel facilities for terminal I/O and mutex locking. This meant that those features could not be debugged using the debugger, and a logic analyzer was the only tool which could be used to debug system hangs in those portions of the kernel. A better implementation would have been as an independent program using the minimum possible kernel functionality.

3. Interrupts

One of the most difficult aspects of dealing with multithreading SVR4 was the avoidance of deadlocks due to interrupt routine interactions. Because of these interactions, the earliest versions of our kernel required large portions of the code to run with disabled interrupts in order to avoid deadlock. From analyzing these problem situations, the following necessary condition to avoid deadlock was discovered:

- Let A be a lock which is acquired by one or more interrupt routines at a maximum interrupt priority of X.
- Let B be a lock which is acquired while holding A, not necessarily in an interrupt routine.
- Then lock B must ALWAYS be acquired at priority X or higher.

The proof that this condition is necessary to avoid deadlock is by contradiction. Assume that lock B is acquired by a processor at priority lower than X. Then consider that an interrupt at priority X occurs on this processor which tries to acquire lock A. Further, assume that another processor holds lock A and tries to acquire lock B at the same time. This situation results in deadlock. Therefore, it is a necessary (but not sufficient) condition to avoid deadlock that any time lock B is held, any interrupt that might try to acquire lock A must be delayed by holding lock B at priority X or higher.

This condition is particularly annoying because it is not necessary for a lock to actually be acquired at interrupt level to require protection. In pre-SVR4 kernels, the typical multithreading strategy to avoid this type of deadlock was to never acquire any lock B while holding lock A. This is generally not possible in SVR4 due to the fact that it is allowable to call the main kernel memory allocator, *kmem_alloc*, from an interrupt routine. The memory allocator can be called either directly or indirectly through the *STREAMS* interface. The original version of this function required a lock under which were nested most of the virtual memory subsystem locks. Because of the requirement above, the entire virtual memory subsystem was required to run with interrupts disabled. The fix for the problem involved allocating memory from a special interrupt-only memory pool, thus bypassing the offending lock. As it turns out, allocation calls from interrupts are not the only problem area. If a lock in a driver is acquired both at system call and interrupt level, then *kmem_alloc* from a driver function holding the lock must also allocate from the interrupt-only pool.

4. Translation Look-Aside Buffer Shootdown

One of the problems related to the multithreading of the virtual memory portion of the kernel is the maintenance of translation look-aside buffers (TLBs) on each processor in the system. Cache consistency for regular main memory caches is assumed to be provided by the hardware in a multiprocessor system. However, hardware consistency of address translation caches within a microprocessor is usually not provided. It may be necessary for the kernel to invalidate TLBs across all of the CPUs in the system when a virtual memory mapping is changed by modifying a page table entry.

This TLB invalidation operation, which has been dubbed a TLB shootdown or global TLB flush, can be very expensive. The reason is that some processor types may operate incorrectly when a page table that they are currently using is changed, particularly if the translation is invalidated. In this case, performing a TLB shootdown requires that every processor in the system be put into an idle state while any globally accessible page table entry is modified. So, the general algorithm for a global TLB flush is as follows:

— Interrupt all CPUs

— All CPUs flush TLBs and wait

— Modify page table entries

— Release CPUs

This algorithm is costly and does not scale well with the number of processors. Although this approach is safe and guarantees no accesses through stale TLB entries, it is generally unnecessary as addresses are not usually being used by all of the processors in the system. In the Intel family, local flushing of a CPU's entire TLB cache is done on each context switch, and hence any given process' user-mode page table entries can only be cached on a single CPU which is running that process. (Note that this property will likely disappear when support for threads sharing a single address space is added to the kernel.) However, even without threads, system virtual address mappings and their page tables are shared across all processors in a system.

One might expect this not to be a major problem in that system virtual translations ought not to change that often. While this was true in pre-SVR4 UNIX versions to a large extent, there is a major change in SVR4 which increases the rate at which system translations are changed. This change involves the way in which file access is done. SVR4 uses the virtual memory approach described by Gingell [7] where file I/O is done through the *segmap* segment driver. Each file access can result in a virtual mapping change to make a portion of a file visible through a virtual window.

A very good discussion of this algorithm in the context of Mach is given in Black *et al.* [3]. Mach's virtual memory system maintains machine-dependent physical mapping (*pmap*) information for virtual address spaces used on each processor. The pmap contains information about which CPUs are using the address space, and hence the TLB flush operation can be confined to the subset of processors actually using the mapping being changed. The subset of processors using a pmap is generated on demand, in a manner which Black calls ''lazy evaluation''. Their performance analysis leads them to conclude that in the context of a few hundred processors, user TLB shootdowns will not present performance problems, but that kernel shootdowns might, estimating a 10% overhead on such a machine. In the SVR4 case, a fairly simple stress test easily generated over 200 kernel TLB flushes per second on a single processor [12], much higher than the rates implied by the tests described by Black. The test included one disk bound program ("ls -R /") and one fork/exec bound application. At 500 microseconds per flush, which Black measured as the time taken for 2 CPUs, this amounts to an estimated 10% overhead for 2 processors, rather than 200 processors. Although many of these flushes are not required to be global TLB flushes, these numbers provided the motivation for the development of yet another ''Lazy'' TLB Flushing approach.

4.1 Lazy TLB Flushing

The design of the lazy TLB flushing algorithm was motivated by the characteristics of the SVR4 TLB flushing strategy for the Intel processors used in the development base. The key point of this strategy is that the entire TLB is flushed during a context switch or an explicit local TLB flush. This means that, in fact, stale TLB entries tend not to last very long. This is different from a software-loaded TLB with address-space tags where TLB entries can persist across context-switches. A good example of another lazy strategy (dubbed ''lazy devaluation'') which deals with this type of cache in a manner which has similarities to ours is described by Thompson *et al.* [17].

The concept behind the algorithm is to tag each TLB flush and virtual address invalidation with the value of a generation counter. A global generation counter is incremented by each *local* TLB flush operation or context switch and a copy of the counter is saved by the processor doing the flush as its per-CPU generation count. When a kernel virtual address is invalidated and freed, the address is tagged with the current value of the generation count. When that address is reallocated, its tag is compared with all of the per-CPU counts. If any CPU has a lower generation count, then a global TLB flush is performed. Otherwise, all of the CPUs would have done a local flush since the address was freed, and hence no stale TLB entry can possibly exist.

A major assumption behind this algorithm is that invalidation of a kernel address implies that no references will be made to the address until it is reallocated and bound to a new physical address. As a result, stale TLB entries should never be referenced, and are thus safe. This condition is easily maintained when the address in question is controlled by one of the SVR4 segment drivers. A segment driver which uses the lazy TLB flush algorithm is responsible for maintaining the generation count in a structure associated with each virtual address used by the segment. In addition, to achieve the full benefit of the algorithm, segment addresses should be freed and reallocated in first-in, first-out order. Doing this increases the time between freeing an address and reallocating it, so that the probability of flushing a stale TLB entry is increased also.

As an example of how well this algorithm reduces the number of global TLB flushes where it is used, a sample from a running SVR4MP system with 2 CPUs showed that out of 217849 potential global flushes due to address reallocation, 71 actually occurred.

A second source of a potentially large number of global TLB flushes is the page cleaning daemon. The page reclaim algorithm could require a flush for each page scanned for aging or freeing, since the scan involves manipulating bits in the page table entry. Reduction of flushing in this context was accomplished by batching pages for processing and amortizing the global TLB flush algorithm over a number of page invalidations.

5. Cache Locking

One of the primary techniques used in UNIX to enhance performance is the use of numerous caches for important types of data. For example, until recent versions, all file system I/O in UNIX went through a disk buffer cache, which represented copies of disk blocks which were currently held in main memory.

The logical structures used to implement these caches tend to have a number of features in common:

- Cache objects are maintained on two separate queues, a hash queue and a free list.
- The search for an object in the cache is done through a hashing function which maps the object identifier to a hash queue, which is searched for a cache entry containing the desired object.[1]
- When a search results in a cache hit, the entry found is typically marked busy, removed from the free list, and returned to the caller. When finished with the entry, the caller invokes a function which marks the entry non-busy and returns it to the free list, usually at the end in order to maintain LRU ordering. When the caller wants to invalidate the entry, it may be placed on the front of the free list.
- Cache misses are dealt with by removing the first cache element from the free list, possibly performing some invalidation function, removing the entry from its old hash list, and initializing and adding it to the hash list appropriate for the desired object.
- Cache flushing or copy-back is accomplished by a periodic scan through the entire free list looking for cache entries that need to be cleaned, e.g., by being written to disk. The cache entry is removed from the free list, cleaned and replaced on the free list. This flushing is done so that cleaned entries are available for reassignment when a cache miss occurs.

Figure 2 shows a simplified code template of cache lookup and release functions which illustrate the properties listed. Notice that the entire cache is protected by a single mutex lock, *cache_lock*, which represents a possible bottleneck in the system. Also notice that the multithreading literally involves adding the MUTEX_LOCK and MUTEX_UNLOCK calls around the main bodies of the functions.

Usually, caches are configured to have fixed, or at least bounded, size. It is possible to do away with the free list and use the general kernel memory allocator to create new cache entries whenever a miss occurs, and somehow reclaim stale entries. This has been done for the inode cache [2] and for many kernel data structures [14]. However, there must still be some limit to avoid consuming too much memory for a given cache. This implies that there is still a need for some type of free list from which entries can be reclaimed.

5.1 Queues Considered Harmful

The problem with such free list queues is that as the system is scaled up, the queue size tends to increase linearly with the size of memory. Any operation of the form *for some x on a free list do a(x)* presents significant problems if *a(x)* allows the queue to change concurrently. Such operations have

1. Historically, the typical hash function used in such caches involves generating an index to an array of hash queue heads by taking f(id) modulo 2^k, where f(id) is some function of the object identifier *id* and *k* is such that the number of hash buckets is reasonable. This was done originally so that the modulus operation could be simplified to f(id) & $(2^k - 1)$ on the PDP11, since a general modulus involved calling a software divide subroutine. It turns out that many of the caches have object identifiers that are always separated by powers of 2, so that some hash queue heads can never have anything on them. A prime number close to the desired size would be the best choice for the divisor in the modulus operation, although using $2^k - 1$ should provide much better distribution of hash indices than the original approach.

```
/* Cache Lookup and Release Functions */

struct cache_entry {
        struct cache_entry *hash_next, *hash_prev; /* Hash list */
        struct cache_entry *free_next, *free_prev; /* Free list */
        int id;                                 /* Object id */
        int flags;                              /* Status flags */
        ...                                     /* Data associated with id */
};

#define NUMHASH 59          /* Use odd number */
struct cache_entry hashqs[NUMHASH];         /* Only hash_next and hash_prev used */
struct cache_entry freelist;                /* Only free_next and free_prev used */
mutex_t cache_lock;                         /* Global lock for cache */

/* Look for object with tag object_id in the cache */
struct cache_entry *
getcache(object_id)
int object_id;
{
        register struct cache_entry *hashq, *qp;

        hashq = &hashqs[object_id % NUMHASH];
        MUTEX_LOCK(&cache_lock);
again:
        for (qp = hashq->hash_next; qp != hashq; qp = qp->hash_next)
                if (qp->id == object_id)
                        break;
        if (qp != hashq) {                  /* Cache Hit */
                if (qp->flags & BUSY) {
                        sleep(qp, PCACHE);
                        goto again;         /* In case the cache changed */
                }
                free_remove(qp);
        } else {                    /* Cache Miss */
                qp = freelist.free_next;
                free_remove(qp);
                hash_remove(qp);
                ...                         /* Destroy old contents */
                qp->id = object_id;
                ...                         /* Set up the rest of the fields */
                hash_add(hashq, qp);
        }
        qp->flags |= BUSY;
        MUTEX_UNLOCK(&cache_lock);
        return qp;
}

/* Return cache entry obtained with getcache */
putcache(qp, flag)
struct cache_entry *qp;
{
        MUTEX_LOCK(&cache_lock);
        if (flag == KEEP)
                free_add(freelist.free_prev, qp); /* At End */
        else
                free_add(&freelist, qp);          /* At Front */
        qp->flags &= ~BUSY;
        MUTEX_UNLOCK(&cache_lock);
        wakeup(qp);                         /* In case qp is wanted */
}

/* Clean dirty entries in cache - called periodically */
flushcache()
{
        register struct cache_entry *qp;

        MUTEX_LOCK(&cache_lock);
again:
        for (qp = freelist.next; qp != &freelist; qp = qp->next)
                if (NEED_FLUSH(qp)) {
                        free_remove(qp);
                        qp->flags |= BUSY;
                        MUTEX_UNLOCK(&cache_lock);
                        doflush(qp);        /* Write to disk, etc. */
                        MUTEX_LOCK(&cache_lock);
                        free_add(&freelist, qp);
                        qp->flags &= ~BUSY;
                        wakeup(qp);
                        goto again;         /* Queue may have changed */
                }
        MUTEX_UNLOCK(&cache_lock);
}
```

Figure 2: Cache Lookup and Release Sample Code

been coded so that after each *a(x)* operation, the queue is rescanned from the beginning, which is typically O(N^2), where N is the size of the free queue. The prime example of this behavior is the buffer cache flushing algorithm, which performs the copy-back function for modified blocks in the cache. A System V Release 3 machine with a large cache has been seen to take 30 continuous seconds of CPU time to perform the cache flush using this algorithm. During this time no other processing was accomplished.

One solution to this type of problem, which has been used several places in SVR4MP, is to batch the elements requiring the *a(x)* operation by collecting them on a separate "To Do" list. This can be done in one scan through the free list queue, followed by the application of *a(x)* for each element on the "To Do" list. This reduces the algorithmic complexity of the operation to O(N).

This problem illustrates a general principle which needs to be followed for reasonable scalability of a multiprocessor kernel, namely that queue sizes should remain bounded by a constant as the system size is increased. Even the O(N) behavior possible using the "To Do" list will become a bottleneck since the scanning of a large, single queue can not be easily parallelized to use more than one processor. One solution to this problem is to restructure the single logical queue into a number of smaller queue segments, each of which could be processed in parallel by a separate processor.

5.2 Software Set-Associative Caches

The multiprocessor locking required for caches protects the hash and free list queues and makes object identity changes atomic with respect to cache searches. Most implementations of locking such caches require obtaining a common lock which protects the free list during both lookup and release operations. This locking tends to constrain throughput with a larger number of processors. As an example, consider the file system buffer cache. Attempts to remove the bottleneck by using separate locks on hash and free lists have generally not been successful, as the locks on the hash list do nothing to alleviate contention on the single free list lock. Indeed, the additional overhead and complexity of dealing with multiple hash queue locks have been found to lower overall buffer cache throughput [4, 9, 13]. Clearly, for better scaling, there need to be multiple locks for the free queue, which means splitting the free list into numerous queues. One way to do this is to have a free list queue for each hash queue, so that the hash queue lock would also protect a free list of all non-busy elements on that hash queue. If a cache miss is satisfied by reallocating a cache element from the searched hash queue, the entire operation can be done by holding only one lock, which protects only a hash queue's worth of the cache. In addition, hash dequeueing and requeueing operations are no longer necessary, since the new identity of the element belongs on the same hash queue.

In the case where the cache contains a fixed number of statically-allocated elements, the organization can be thought of as a 2-dimensional array which very much resembles the organization of a hardware set-associative cache. For this reason, we decided to name this organization a *Software Set-Associative Cache*. If an actual array is used to implement the data structure, the queue pointers become unnecessary, although maintaining an LRU free list might be easiest with an actual queue.

Figure 3 shows the code for a sample cache with the new organization, modified from Figure 2. The major difference is that the mutex lock associated with the appropriate *hashqs* structure is locked, removing the previous bottleneck due to *cache_lock*. Up to *NUMHASH* cache operations can be done in parallel, where *NUMHASH* is configured so that the average length of each hash queue stays constant as the number of cache entries is increased. The *flushcache* contention is reduced since only a single hash queue's free list is locked and processed at once. The flush operation is still O(N^2), but N in this case is a small constant, namely the size of the hash queue.

While tuning SVR4MP on 2 CPUs, it was found that the cache used to hold pages during file system operations, the *segmap* cache, experienced lock contention between 10 and 15 percent using a single lock for the entire segmap cache. With one day's effort, the cache was reorganized as a software set-associative cache using the existing hash queues and adding a free list queue head to each hash queue head. The hash queue manipulation routines actually simplified to almost nothing, making the code more efficient. The measured contention on the hash queue class of locks was found to have decreased

```
/* Software Set-Associative Cache Lookup and Release Functions */

struct cache_entry hashqs[NUMHASH];
mutex_t hash_locks[NUMHASH];            /* Mutex locks */

/* Look for object with tag object_id in the cache */
struct cache_entry *
getcache(object_id)
int object_id;
{
        register struct cache_entry *hashq, *qp;

        hashq = &hashqs[object_id % NUMHASH];
        MUTEX_LOCK(&hash_locks[object_id % NUMHASH]);
again:
        for (qp = hashq->hash_next; qp != hashq; qp = qp->hash_next)
                if (qp->id == object_id)
                        break;
        if (qp != hashq) {                      /* Cache Hit */
                if (qp->flags & BUSY) {
                        sleep(qp, PCACHE);
                        goto again;             /* In case the cache changed */
                }
                free_remove(qp);
        } else {                /* Cache Miss */
                qp = hashq->free_next;
                free_remove(qp);
                ...                             /* Destroy old contents */
                qp->id = object_id;
                ...                             /* Set up the rest of the fields */
        }
        qp->flags |= BUSY;
        MUTEX_UNLOCK(&hash_locks[object_id % NUMHASH]);
        return qp;
}

/* Return cache entry obtained with getcache */
putcache(qp, flag)
struct cache_entry *qp;
{
        hashq = &hashqs[qp->id % NUMHASH];
        MUTEX_LOCK(&hash_locks[qp->id % NUMHASH]);
        if (flag == KEEP)
                free_add(hashq->free_prev, qp);
        else
                free_add(hashq, qp);
        qp->flags &= ~BUSY;
        MUTEX_UNLOCK(&hash_locks[qp->id % NUMHASH]);
        wakeup(qp);
}

/* Clean dirty entries in cache - called periodically */
flushcache()
{
        register struct cache_entry *hashq, *qp;
        register int i;

        for (i = 0; i < NUMHASH; i++) {
                hashq = &hashqs[i];
                MUTEX_LOCK(&hash_locks[i]);
again:
                for (qp = hashq->next; qp != hashq; qp = qp->next)
                        if (NEED_FLUSH(qp)) {
                                free_remove(qp);
                                qp->flags |= BUSY;
                                MUTEX_UNLOCK(&hash_locks[i]);
                                doflush(qp);            /* Write to disk, etc. */
                                MUTEX_LOCK(&hash_locks[i]);
                                free_add(hashq, qp);
                                qp->flags &= ~BUSY;
                                wakeup(qp);
                                goto again;
                        }
                MUTEX_UNLOCK(&hash_lock[i]);
        }
}
```

Figure 3: Software Set-Associative Cache Sample Code

to less than .1 percent. The cache was configured statically with 4 cache elements in each hash queue. The beauty of this organization is that as the cache size is increased with more memory and more processors, the number of hash queues and locks also increases, so that contention should not increase and maximum throughput should increase linearly with cache size.

One of the bonus side-effects of this organization is that because contention is so low, the hash queue locks can also be used to protect non-lookup operations on individual cache entries. This removes the need for a separate lock inside each cache entry to protect its internal data fields. In the segmap cache, the hash queue lock was used to protect individual segmap entries during fault operations.

This reorganization is general enough to apply to a number of other kernel caches, such as the pathname lookup cache and the buffer cache. When this technique was applied to both of these caches, contention reductions of similar magnitude to the segmap cache were obtained.

6. Performance Measurements

The main performance goal for SVR4MP involved the scalability of the system, that is, the proportional increase in system throughput when additional processors are brought online. The (somewhat arbitrary) goal of the project was that each additional processor deliver at least 85% of the throughput of the first processor. The reported results were obtained using the GAEDE benchmark [6] running on a Wyse 9000i multiprocessor system.

6.1 Hardware Configuration

The Wyse 9000i was configured with five 25-MHz 80486 microprocessors (each with 8 Kbyte of on-chip cache and 128 Kbyte of external copy-back cache), 64 Mbyte of RAM, 2 SCSI Peripheral Adaptors with 2 SCSI channels per adaptor for a total of 4 SCSI channels, two disks per SCSI channel for a total of eight disks, a Wyse terminal and a keyboard. One of the drives was used as a swap device, while the other 7 drives were configured with UFS-4K file systems, which are derived from the Berkeley Fast File System via SunOS [8] and use a 4-Kbyte block size.

6.2 Benchmark Results

The GAEDE benchmark is a multi-user benchmark that estimates system throughput by measuring the average completion time of a certain number of shell scripts running concurrently. Each script simulates a typical Unix development session doing `nroff`, `cc`, `cpio`, `find`, etc. on different working directories. Eight working directories, spread over 7 disks, are used. They are clobbered at the start of each run (to remove files left over from previous runs) and re-populated before the actual benchmark run. Results were collected for 5, 10, 15, 20, 25 and 30 scripts, repeated 5 times to obtain averages. Table 1 lists the GAEDE throughput and scalability as measured on the Wyse 9000i. In all cases, the average I/O waiting time and idle time were measured to be less than 5%.

Gaede Throughput in Processes / Hour						
CPUs	Scripts					
	5	10	15	20	25	30
1	43400	44095	42296	41348	40479	39984
2	76501	82942	81909	81758	80514	79480
3	99195	115031	117736	117721	116255	115180
4	108418	143003	148589	148663	148006	147662
5	110882	163696	171351	175228	174641	175586
Scalability						
CPUs	Scripts					
	5	10	15	20	25	30
1	1	1	1	1	1	1
2	1.76	1.88	1.93	1.97	1.98	1.98
3	2.28	2.60	2.78	2.84	2.87	2.88
4	2.49	3.24	3.51	3.59	3.65	3.69
5	2.55	3.71	4.05	4.23	4.31	4.39

Table 1: SVR4MP Scalability Results

The larger the number of scripts, the higher the total throughput. From the total throughput perspective, 5 processors deliver 4.39 times the throughput of the first processor when 30 scripts are running. This satisfies the 85% scalability requirement for up to 5 CPUs. The kernel, however, may not scale as well beyond the 5th processor. Starting with the first processor, each additional processor brought online delivers 0.98, 0.90, 0.81 and 0.70 times the throughput of the first processor. From this perspective, the incremental increase in throughput per additional processor fell below the 85% requirement when there are 4 or more processors. Had the platform been capable of supporting a sixth processor, it would probably deliver no more than 0.70 times the throughput of the first processor.

It should be noted that the scalability is a property of the *combination* of hardware and software used for these tests. It may well be that the memory subsystem of the Wyse system starts to become saturated at 5 processors, and that the decrease in incremental throughput is really due more to this rather than increased locking contention in the software.

7. Summary

While modifying SVR4 to run on a multiprocessor, a number of interesting difficulties were encountered and overcome. The locking model extensions of recursion and freeing mutex locks across context switches were very successful in simplifying the code changes required to multithread the kernel. In fact, comparisons of the modified sources with the original show that a very large percentage of the changes involve only the addition of mutex lock and unlock calls around critical code sections. The lock debugging tools were indispensable in the analysis and removal of deadlocks and bottlenecks.

It has been interesting watching the evolutionary steps that a number of kernel components have gone through in arriving at their current states. The programmers involved in the project all had varying degrees of multiprocessor experience, and it may be worthwhile describing the types of mistakes made along the way.

The most elementary type of error that was made involved under-locking the true scope of a critical section. Some naive locking which protected certain shared variables literally placed a lock and unlock around all reads or writes of the variables involved, without considering that some amount of code after the read of a variable might rely on the value staying constant.

The second common error was defining locks at too fine a level. There are several difficulties with this. First, in order to avoid deadlock, it was often necessary to unlock one lock type before locking another, and then reversing the process. This not only affected the robustness of the code, since the unlocking of the first lock could open windows for races to occur, but meant that the rate of locking and unlocking was very high. In an early version of the STREAMS TCP/IP networking implementation, the processing of a single packet transaction was seen to take 80 separate lock operations. Races caused by unlocking and relocking caused a number of problems, particularly unserialized system calls and out-of-sequence data. Secondly, with very fine levels of locking, the lock overhead tends to dominate the processing time. As a consequence, the packet throughput degraded from the uniprocessor kernel and no significant performance was gained by using multiple CPUs.

As another example, early locking of the buffer cache used a single global lock to protect the hash and free list data structures, and a separate lock inside each cache entry to lock its fields. It turned out that locking of the per-entry lock was almost always nested inside the global lock, so was actually unnecessary. Removing the per-entry lock completely and using the global lock to protect the internal fields roughly halved the locking overhead for the buffer cache, with an actual decrease in the measured contention (on 2 CPUs). Increasing the number of CPUs increased the contention on this global lock, the solution for which was the Software Set-Associative Cache structure previously described.

The hardest types of errors to find and solve involved some surprisingly subtle critical section violations and deadlocks. They are actually similar in nature to the elementary errors described above, but involve much more complicated and less probable interactions between processors. Unfortunately, because of their complexity, such errors are difficult to find by code inspection unless a system crash has occurred which suggests their presence. Many of these bugs were races which existed in the original uniprocessor code. The additional real concurrency in the multiprocessor version of the system made their manifestation more likely.

Some problems could only be overcome with inelegant solutions due to "features" of the original SVR4 design. It could be argued that providing general memory allocation capability from interrupt routines is not a good idea, particularly in a multiprocessor context. In fact, it could be argued that doing much of anything at interrupt level is not a good idea. Processors have become dramatically faster over the last few years, so the need to avoid a context switch to satisfy real-time requirements is much less important. A model which uses some form of kernel light-weight process (LWP) for interrupt processing allows the LWP to be suspended to wait for locks or events. This type of model has distinct advantages in reducing the time that the kernel must run with interrupts disabled and avoiding multiprocessor deadlocks. In a related vein, the STREAMS interfaces allow coupling between queues in a stream which is similar to interrupts (and which can be called from interrupt routines). This tight coupling of STREAMS queues made it very difficult to provide high concurrency among STREAMS modules because of circular dependencies among queue locking. Ironically, the aspect of the interface which was meant to increase responsiveness by allowing interrupt-like processing prevents an increase in responsiveness which could be achieved in a multiprocessor by processing STREAMS queues in parallel. STREAMS and networking modules are areas of the kernel that require more work to achieve the level of scalability demonstrated for the rest of the system. Indeed, the STREAMS locking protocol has reached at least its third version, and deserves to be the subject of a paper in its own right.

The original goal of this effort was to support somewhere between 8 and 16 processors. A significant number of contention points have been identified and removed with the aim of continually improving the scalability of the kernel. Use of the Software Set-Associative Cache approach on three of the file-system related caches has raised their scalability to be essentially limited only by their size. As of this writing, locks with the highest contention are in the process management and hardware address translation areas. Increasing the scalability of the kernel involves finding queues or other collections of data protected by locks and attempting to enforce two properties: Firstly, the amount of data protected by each lock must not increase as the size of the system memory and configuration parameters are increased. Secondly, the access rate for data protected by a lock must not increase, which typically means increasing the number of locks of a given type as more CPUs are added.

The SVR4MP effort resulted in a system which largely met its goals. The degradation relative to a uniprocessor SVR4 kernel is very close to the 5% target, and the scalability of 5 processors (no hardware with 6 CPUs was available) is above the 85% per processor scalability goal.

SVR4MP is available from UNIX Systems Laboratories as a source-code upgrade to System V Release 4. More information can be obtained by calling 1-800-828-UNIX.

8. Acknowledgments

This project has been a pleasure for us to be involved with, as most of the architecture team had at least two generations of multiprocessor design experience. A large number of people have contributed to the success of SVR4MP, and this list might well be considered a continuation of the list of authors, as most have written code: Mike Abbott, Sunil Bopardikar, Fiorenzo Catteneo, Calvin Chou, Ho Chen, Ben Curry, Jane Ha, Jim Hanko, Mohan Krishnan, John Litvin, Mani Mahalingam, Arun Maheshwari, Cliff Neighbors, Sandeep Nijhawan, Mark Nudelman, Lisa Repka, K. M. Sherif, Moyee Siu, John Slice and Vijaya Verma. Thanks are also due to the reviewers for their helpful suggestions for improvements to this paper.

9. References

[1] M. Bach and S. Buroff, Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 64:1733-1749, October 1984.

[2] R. Barkley and T. P. Lee. A Dynamic File System Inode Allocation and Reclaim Strategy. *Proceedings of the Winter 1990 USENIX Conference.*

[3] D. Black, R. Rashid, D. Golub, C. Hill and R. Baron. Translation Lookaside Buffer Consistency: A Software Approach. *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 1989.

[4] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. *Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pp 105-126, October 1989.

[5] M. Campbell, Richard Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith and R. Wescott. The Parallelization of UNIX System V Release 4.0. *Proceedings of the Winter 1991 USENIX Conference.*

[6] S. Gaede. A Scaling Technique for Comparing Interactive System Capacities. *Proceedings of the Conference of CMG XIII*, December 1982.

[7] R. Gingell, J. Moran and W. Shannon. Virtual Memory Architecture in SunOS. *Proceedings of the Summer 1987 USENIX Conference.*

[8] S. Kleiman. Vnodes: An Architecture for Multiple File Systems in Sun UNIX. *Proceedings of the Summer 1986 USENIX Conference.*

[9] G. Hamilton and D. Conde. An Experimental Symmetric Multiprocessor ULTRIX Kernel. *Proceedings of the Winter 1988 USENIX Conference.*

[10] A. Langerman, J. Boykin and S. LoVerso. A Highly-Parallelized Mach-based Vnode Filesystem. *Proceedings of the Winter 1990 USENIX Conference.*

[11] S. LoVerso, N. Paciorek, A. Langerman and G. Feinberg. The OSF/1 Unix Filesystem (UFS). *Proceedings of the Winter 1991 USENIX Conference.*

[12] M. Nudelman. TLB Flushing in SVR4.0/MP. Internal Memorandum. January, 1990.

[13] K. Peacock. The CounterPoint Fast File System. *Proceedings of the 1988 Winter USENIX Conference.*

[14] R. Rodriguez, M. Koehler, L. Palmer and R. Palmer. A Dynamic UNIX Operating System. *Proceedings of the Summer 1988 USENIX Conference.*

[15] L. M. Ruane. Process Synchronization in the UTS Kernel. *Computing Systems*, Vol. 3 No. 3, USENIX, Summer 1990.

[16] U. Sinkewicz. A Strategy for SMP ULTRIX. *Proceedings of the Summer 1988 USENIX Conference.*

[17] M. Thompson, J. Barton, T. Jermoluk and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. *Proceedings of the 1988 Winter USENIX Conference.*

Dynamic Synchronization of Real-Time Threads for Multiprocessor Systems

Hongyi Zhou hongyi@cc.gatech.edu
Karsten Schwan schwan@cc.gatech.edu

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Ahmed Gheith
gheith@futserv.austin.ibm.com

IBM Advanced Workstation Division
Future Systems Technology
11400 Burnet Rd., Bldg 820, ZIP 2812
Austin, TX 78758

November 5, 1991

Abstract

We study mutual exclusion and synchronization for *dynamic* hard real-time multiprocessor applications. As with any dynamic parallel program, a dynamic real-time application's execution can result in on-line creation of additional tasks, and the creation of such time-constrained tasks cannot be predicted or accounted for prior to program execution. The research results presented in this paper concern task synchronization such that guarantees can be made regarding the synchronized tasks' timing constraints. Such guarantees cannot be made without performing on-line schedulability analysis and on-line analysis concerning the maximum time that a task will wait for some resource being acquired with a synchronization primitive. We present a real-time locking scheme that prevents deadlocks and ensures time-bounded mutual exclusion. The maximum waiting time for a task attempting to acquire a resource is computed with an $O(1)$ algorithm. Two important attributes of the algorithm are: (1) previously made guarantees regarding resource accesses are always maintained, and (2) failures regarding accesses to shared resources are reported immediately. As a result, the application program or higher-level operating system software can deal with such failures in a timely manner, by acquisition of alternative resources, by execution of exception handling code, etc. The algorithm for waiting time determination and the synchronization primitive using it are part of a real-time, multiprocessor threads package implemented on standard Unix platforms and on a 32-node BBN Butterfly multiprocessor.

1 Introduction

Synchronization primitives for multiprocessor systems have been of significant interest to system architects, operating system developers, and application programmers ever since multiprocessor technology was first developed. While the implementation of locking primitives, of semaphores, and of other mechanisms for synchronization is fairly well understood for the different classes of parallel machines, specific synchronization requirements imposed by different application domains are only recently being investigated. The application domain driving the work on thread synchronization reported in this paper is that of time-bounded systems (e.g., multi-media systems[15], telecommunication protocols used with time-bounded applications (e.g., Bellcore's Touring Machine project), classical real-time applications in robotics and manufacturing, etc.). In such systems, arbitrary delays due to task synchronization cannot be tolerated. Therefore, for the synchronization primitives used by real-time programs, this implies that such waiting times must be specified, analyzed, and enforced in conjunction with their use.

Past work on synchronization for real-time systems has focussed on the effects of mutual exclusion requirements on shared resources when scheduling a *statically* defined set of periodic tasks with deadlines[13], typically using priority-based scheduling techniques[20]. In this paper, we study mutual exclusion and synchronization for *dynamic* hard real-time multiprocessor applications. A real-time application is *dynamic* if its execution can result in the creation of time-constrained tasks that cannot be predicted or accounted for prior to program execution. Sample dynamic tasks in actual real-time applications include planning and exception handling tasks for robotics applications[18, 1, 2, 8, 6], software components that make higher level decisions in intelligent autonomous systems[1], multi-media applications built to interact with end users imposing varying demands, telecommunication protocols , and similar medium to large grain computations in many other domains.

Dynamic real-time applications are interesting to study in part because they impose requirements on the real-time operating system's synchronization primitives that are much like the requirements imposed on multiprogrammed non-real-time multiprocessor systems:

- dynamic arrivals and departures of tasks that may interact with previously created tasks in an arbitrary fashion, including the unforeseen use of shared resources; and
- unpredictable distributions of shared resource accesses and usage.

However, a marked contrast between real-time and non-real-time multiprocessor applications is that significant additional information can be assumed known about each dynamic real-time task, including being periodic (e.g., transmission tasks in continuous media applications) or sporadic (e.g., high-level planning tasks in robotics applications), and its deadline and maximum execution time.

The research results presented in this paper concern task synchronization such that guarantees may be made regarding the synchronized tasks' timing constraints. Such guarantees cannot be made without performing on-line schedulability analysis (i.e., the determination that a task can or cannot meet its deadline) and on-line analyses concerning the maximum time that a task will wait for some resource being acquired with a synchronization primitive. Furthermore, since timing constraints cannot always be met, it must be possible for such schedulability analysis to result in failures regarding task access to shared resources. In this paper, we present a real-time locking scheme that prevents deadlocks and ensures time-bounded mutual exclusion. The maximum waiting time for a task attempting to acquire a resource is computed with an $O(1)$ algorithm, and the maximum time a task will hold an acquired resource is specified by the application programmer (much like real-time programmers must specify or determine maximum task execution times). Two important attributes of the algorithm presented in this paper are: (1) previously made guarantees regarding resource accesses are always maintained, and (2) failures regarding accesses to shared resources are reported immediately. As a result, the application program or higher-level operating system software[6] can deal with such failures in a timely manner, by acquisition of alternative resources, by execution of exception handling code, etc.

The algorithm for waiting time determination and the synchronization primitive using it are part of a real-time, multiprocessor threads package[19] implemented on standard Unix platforms and on a 32-node BBN Butterfly multiprocessor.

The remainder of this paper is organized as follows. The next section discusses the nature of real-time threads as well as the dynamic synchronization problem in multiprocessor real-time systems. Section 3 contains a brief overview of the real-time, multiprocessor threads package developed in our work, mainly focusing on the package's notion of real-time locks. Section 4 presents the algorithm for computing the waiting time for a real-time lock. Section 5 gives experimental results regarding real-time locking. We conclude the paper in Section 6. Issues regarding nested locks and deadlocks are discussed in the appendix.

2 Problem Definition and Related Work

The scheduling of tasks with hard deadlines has been an important area of research in real-time systems. Both non-preemptive and preemptive scheduling algorithms have been studied in the literature[5, 9, 11, 12, 14, 21, 22]. An important problem that arises in the context of such real-time systems is the effect of blocking caused by the need for synchronization among tasks that require exclusive access to shared logical or physical resources. Mok[13] showed that the addition of mutual exclusion requirements in real-time programs makes the general scheduling problem an NP-hard problem.

For uniprocessor systems running periodic tasks, two recent protocols provide effective solutions to the scheduling problem with resource sharing. They are the *kernelized monitor* protocol[13] and the *priority ceiling* protocol[20]. In the kernelized monitor protocol, the *earliest deadline first* scheduling policy is used for task scheduling. All executions in critical sections are nonpreemptable. However, schedulability analysis performed in this protocol requires the use of upper bounds on the execution times of all critical sections appearing in tasks. Since such upper bounds may be overly pessimistic, the use of the kernelized monitor protocol may result in low processor utilization.

The priority ceiling protocol is designed for systems where each task has a fixed priority and the *rate monotonic* scheduling algorithm is used. With this protocol, Sha et al.[20] showed that in the worst case, each task only has to wait for at most one lower priority task to finish in a critical section, and deadlocks cannot occur. However, the priority ceiling protocol cannot be directly used when priorities are dynamic. This is addressed by the recent work of M. Chen et al.[3] who have extended the original priority ceiling protocol to one able to handle dynamic priorities.

K. Jeffay[10] has developed schedulability conditions for a set of sporadic tasks that each consist of a sequence of *phases* each of which may require access to at most one shared resource. In his analysis, tasks' timing constraints as well as resource requirements are assumed known beforehand.

Predictable synchronization on multiprocessor real-time systems offers a new challenge. R. Rajkumar et al.[16] have extended their priority ceiling protocols to multiprocessors. In the extension, they assumed that tasks are statically bound to processors and that the rate monotonic scheduling algorithm is used on each processor.

In our work, tasks are represented as execution threads, where all threads on a given processor share the same address space. Each thread has its own execution environment described by its program counter, stack and hardware registers. Each thread is also characterized by (A, S, C, D) where A is its arrival time, S is the earliest possible time at which its execution may begin (start time), C is the maximum computation time, and D is the deadline by which it must complete its execution, all of which need not be known until the time of thread creation. Threads can be periodic or sporadic. A *sporadic* thread is fully described by (A, S, C, D). For each *periodic* thread, a (A, C, P) describes its arrival time, computation time, and period. Given this tuple, for each of its periods, additional tuples (A, S, C, D) may be derived.

This paper does not describe the dynamic preemptive thread scheduling algorithm used for real-time thread scheduling [22]. Instead, we focus on dynamic synchronization among real-time threads. Specifically, we wish to guarantee the timely access to resources shared among threads, which may require that schedulability analysis is performed for *each resource access*, resulting in 'acceptance' or 'rejection' of such an access. Acceptance implies that a thread can wait on the shared resource, since schedulability analysis shows that its deadline can still be met after it waits for some amount of time specified by the time-out parameter attached to the synchronization call. Rejection implies that the deadline cannot be met and requires that the thread take alternative actions, thereby avoiding useless waiting. In addition, acceptance also implies that the total time spent by the thread within the critical section protected by a real-time lock is bounded. Therefore, we assume that a thread cannot be preempted inside a critical section, unless it tries to acquire additional locks. We will address this issue further when discussing nested locks.

3 A Real-Time Multiprocessor Threads Package

Threads are widely accepted as basic computational entities for uniprocessor and multiprocessor programs. Threads separate the notion of a sequential execution stream from the other aspects of traditional UNIX-like processes such as address space and I/O engagement. This separation of concerns yields a significant performance advantage: thread management operations such as *fork*, *wait on a condition*, *signal* typically require an order of magnitude less time than the corresponding operations on UNIX-like processes.

The *real-time threads package* developed by our group is the basis for the construction of predictable and efficient parallel real-time multiprocessor operating systems[7]. The package has been implemented on standard Unix platforms and on a 32-node BBN Butterfly multiprocessor. Its implementation and interface evolved from a Mach cthreads[4] compatible library[17], which distributes its required functionality and data structures across the nodes of the target parallel machine. On the BBN Butterfly, the threads package maintains on each node (1) a pool of stacks for use by locally executing threads, (2) pools of thread descriptor and timing information blocks, (3) a local ready list in earliest deadline first order – termed EL, for earliest deadline list, (4) other structures used for maintaining scheduling information, and (5) a memory pool for future memory allocation/deallocation requests. Thread assignment to nodes is not currently automated in the low-level threads package. Instead, all thread creation calls explicitly specify a node on which the newly created thread is to be executed.

To allow its use in hard real-time systems, the real-time thread package guarantees the schedulability of each thread at the time of its creation, given that thread deadlines, start times, and maximum execution times are known at that time. Such guarantees are maintained when threads communicate with each other, or synchronize their accesses to shared resources.

The main functionality of the real-time threads package may be divided into calls regarding thread manipulation (e.g., creation), thread control (e.g., synchronization), memory allocation, and higher level utilities (e.g., condition variables). The calls for thread manipulation include the `RTthread_fork` call:

```
RESULT
RTthread_fork(func, arg, node, type, starttime, runtime, deadline)
int             (*func)();
any_t           arg;
int             node, type;
int             starttime, runtime, deadline;
```

which creates a sporadic thread with the specified `starttime`, `runtime`, and `deadline`. The dynamic scheduler must verify the schedulability of the thread being created. The new thread's stack is created only if schedulability analysis shows that the desired deadline for execution can be met.

Two additional thread creation calls allow the creation of periodic and event-driven threads. For a periodic thread, the deadline of each instance is calculated based on its period and its estimated execution time. Similarly, for an event driven thread, its schedulability analysis is done based on its execution time, its deadline whenever the thread is activated due to the occurrence of the event causing its activation, and its maximum frequency of arrival of the event.

For thread control, we are offering the calls `RTthread_sleep`, `RTthread_wakeup`, and `RTlock`. In contrast to non-real-time thread packages, a time-out parameter is attached to each of these calls. Such parameters are used to perform the schedulability analysis for these calls.

As stated above, the `RTlock` is the primary focus of this paper. Thread synchronization is performed with mutex locks. To avoid unmatched lock and unlock instructions and to be able to make guarantees that locks will not be held longer than some maximum amount of time, the only primitive offered by the package is the real-time lock `RTlock`:

```
RESULT
RTlock (rtlock, maxwait, time, func, arg)
LOCK    rtlock;
int     maxwait, time;
int     (*func)();
any_t   arg;
```

This call is used to acquire a binary lock `rtlock`. Once this lock has been acquired, the function `func` with argument `arg` is executed within the critical section represented by that lock. `Rtlock` is unlocked implicitly after function execution. Since we cannot allow threads to wait on locks past some well-defined deadlines, schedulability analysis is performed for the `RTlock` call using the parameter `maxwait` denoting the maximum time the call issuing thread can wait for lock acquisition, and the parameter `time` denoting the maximum execution time within the critical section.

When `RTlock` is called, a successful acquisition of `rtlock`, followed by the execution of `func` with parameters `arg`, followed by the release of `rtlock` results in a successful return only if the entire sequence has been completed in time `time`. In case `rtlock` is locked at the time of the call, schedulability analysis is performed to determine whether the calling thread will be able to acquire the lock within `maxwait` time and if so, whether it will still meet its deadline. Such analysis is made according to the current scheduling information and the calling thread's waiting time for the lock. Specifically, the `RTlock` implementation first executes the algorithm for determining the maximum lock waiting time, presented in the next section. Once this waiting time has been computed, the calling thread's start time is changed to the time at which it will end waiting. Next, schedulability analysis is performed by calling the dynamic scheduling algorithm described in [22].

The most complex synchronization primitive offered by the real-time threads package is the real-time analogue of conditions in Hoare monitors. The details of that primitive and the schedulability analysis associated with it are not described in this paper. Additional calls concern yielding the processor, virtual memory management, error handling, configuration control, and others required by the target execution environment. A complete description of the real-time thread package appears in [19].

4 Computing the Waiting Time for Real-Time Locks

Recall that the parameter `maxwait` in the `RTlock` call denotes the maximum time the calling thread is willing to wait for lock acquisition, and the parameter `time` denotes the maximum execution time within the critical section. If `rtlock` is locked at the time of the call, schedulability analysis is performed to assure that the calling thread can wait on the lock and will still meet its deadline. Such analysis also determines whether or not the calling thread will be able to acquire the lock within `maxwait` time. If that is not possible, the `RTlock` call returns with a failure indication, thereby permitting the calling thread to take alternative actions and avoid useless waiting.

The implementation of each real-time lock is straightforward. Namely, each real-time lock has an internal FCFS waiting queue. The FCFS (First Come First Serve) discipline (instead of an earliest deadline first discipline) is used because it ensures that a thread's waiting time computed at the time of its arrival at the lock does not change due to the later arrival of additional threads at the lock. In the next subsection, we describe an algorithm that calculates the maximum waiting time to be experienced by a thread accessing a real-time lock. This algorithm makes use of the queue associated with the real-time lock and the scheduling information available on each processor at the time of the call. We define the *lock holding time* of a thread as the maximum execution time the thread will spend within the critical section protected by a real-time lock.

4.1 Algorithm Description

An algorithm for a static system. We first consider a static system, where threads are not allowed to arrive dynamically. This implies that the waiting time for a real-time lock calculated at the time of the `RTlock` call cannot change later on. Intuitively, this waiting time is equal to the sum of lock holding times of all threads in the lock's waiting queue. However, this intuition is wrong. Even in a static real-time system, when a thread waiting for a lock is awakened by a thread releasing the lock, it may or may not be executed right away. Instead, such a thread becomes 'eligible' to run, but its actual time of execution is determined by its timing constraints as well as the timing constraints of all other threads executing on the same processor. Therefore, when calculating a thread's maximum waiting time at the time it requests a lock, we must also consider the schedules of all processors on which the threads in the lock's waiting queue will execute. Below, we present an algorithm that performs this computation in constant time. For simplicity, the description below will assume that schedulability analysis always results in acceptance of the thread to the real-time lock. In addition, we assume that a thread holding locks cannot be preempted unless it acquires additional locks.

First, consider a single thread – named T_0 – attempting to acquire a real-time lock. In this case, no waiting is necessary and the lock can be acquired immediately. Next, a second thread – named T_1 – attempting to acquire the same lock will have to wait until the lock is released by T_0. Therefore, the waiting time for T_1 is at most the lock holding time of T_0. Before deriving the waiting time for the i-th thread acquiring the same lock, consider the new schedule constructed during the schedulability analysis at the time T_1 issues the `RTlock` call, which we assume is time t_0. Prior to time t_0, T_1 is the thread currently executing on the processor on which it resides, say P_i. Therefore, the schedule on P_i prior to time t_0 is as shown in Figure 1 (a). After schedulability analysis is completed and results in acceptance of T_1's waiting on the lock, T_1 is then removed from its original time slot in the schedule and shifted forward for two reasons: (1) it has to wait for T_0 to release the lock, and (2) other threads with earlier deadlines but later start times compared to T_1 may become ready for execution during its waiting time. This is depicted in Figure 1 (b), where LD_0 is the lock holding time of T_0 and SS_1 is the start time of T_1's next execution. Note that T_1 is ready for execution at the time $t_0 + LD_0$, but that the actual start time of its next execution is SS_1. As we can see, $SS_1 \geq t_0 + LD_0$. As a result, the time slot between $t_0 + LD_0$ and SS_1 may be given to threads with earlier deadlines.

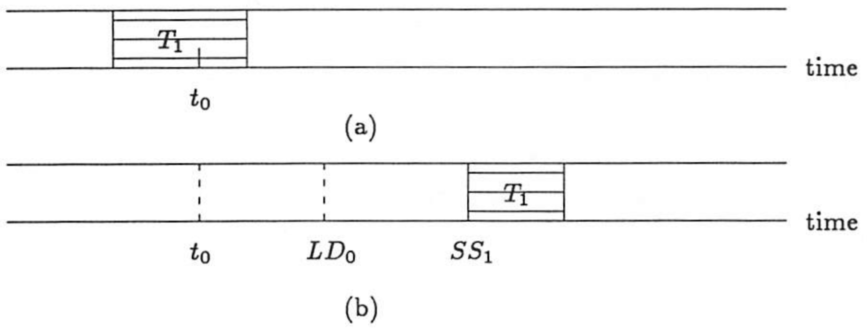


Figure 1: Schedules Before and After a Lock Request

Similarly, for a third thread acquiring the lock, say T_2, its waiting time for the lock is $SS_1 + LD_1 - t_1$, where LD_1 is the lock holding time of T_1 and t_1 is the current time. This waiting time is the same regardless of whether T_1 and T_2 reside on the same processor or on different processors. In general, if T_{i-1} is the last thread in the queue associated with the lock, the waiting time for the i-th thread, T_i, acquiring a lock is $SS_{i-1} + LD_{i-1} - t_{i-1}$, where LD_{i-1} is the lock holding time of T_{i-1}, SS_{i-1} is the start time of T_{i-1}'s next execution, and t_{i-1} is the current time (the time of computing). Note that this computation may be performed in constant time if the start time of a thread's next execution is recorded in its control block whenever a new schedule is constructed.

An algorithm for a dynamic system. Next, we consider dynamic real-time applications. Unfortunately, in this case, the algorithm shown above cannot be generalized trivially. Recall that in a dynamic system, threads can arrive randomly. This implies that a thread's waiting time for a real-time lock computed at the time of lock access may change during the ensuing wait simply because of the arrival of new threads with earlier deadlines (which may cause the construction of new schedules). For instance, in the previous example, T_2's waiting time is $SS_1 + LD_1 - t_1$ at the time T_2 requests the lock. If SS_1, the start time of T_1's next execution, is changed later because a number of new threads with deadlines earlier than that of T_1 are added to the processor on which T_1 resides, then T_2's waiting time has to be updated accordingly. However, the decision regarding whether or not T_2 should be accepted or rejected for this lock was made at the time of lock access, based on old information. As a result, an acceptance 'guaranteeing' timely execution (i.e., the ability to meet a deadline) may become invalid; T_2 may then behave unpredictably. Our solution to this issue is straightforward: we simply reject those to-be-forked threads that jeopardize the timely execution of existing threads currently waiting for locks. As a result, the algorithm described above that computes the waiting time for a real-time lock in static systems is correct for dynamic systems, as well. However, schedulability analysis for to-be-forked threads is complicated since it has to take into account all existing threads waiting for locks. This problem is discussed next.

When determining whether the acceptance of a to-be-forked thread will affect other threads waiting for locks, an obvious time-consuming solution involves the explicit inspection of all waiting queues associated with real-time locks. Our solution is to 'reserve' some time slots for threads currently residing in lock waiting queues to guarantee that they will acquire their locks as scheduled. To clarify, consider the previous example. If we reserve the time slot starting at SS_1 for thread T_1, then the waiting time of T_2 cannot be changed due to new arrivals. In general, for each thread waiting for a real-time lock, we simply reserve the first time slot to which it is assigned for execution at the time of its lock request (recall that scheduling is preemptive). In this fashion, any new arrivals conflicting with threads that are waiting for locks are simply rejected because they becomes unschedulable due to the presence of reserved time slots.

The above scheme is implemented as follows. When some thread T_j on processor P_i has to wait for a real-time lock, a new schedule is constructed for P_i. In the new schedule assuming that t_1 is the start time of the first time slot assigned to T_j in the new schedule, t_2 is the end time of the first slot, and E_1 is the length of this slot, a dummy thread with start time equal to t_1, deadline equal to t_2, and execution time equal to E_1 is created. All dummy threads on the processor are then explicitly considered when new threads' schedulability analysis is performed.

The resulting dynamic locking analysis algorithm can be summarized as follows:

```
RESULT
RTlock(rtlock, maxwait, time, func, arg)
LOCK    rtlock;
int     maxwait, time;
int     (*func)();
any_t   arg;
{
    T = the currently executing thread;
    if (rtlock is available)
    {
      /* a rough outline */
      acquire rtlock;
      r = func(arg);
      release rtlock;
      return(SUCCEED);
    }
    else
    {
      let $T_j$ be the last thread in rtlock's waiting queue;
      SS = the start time of $T_j$'s next execution;
```

```
            LD = the lock holding time of $T_j$;
            waittime = ( SS + LD ) - current time;

            if (waittime > maxwait)
               return(WAITTIME_TOOLONG);
            else
            {
               endwait = current time + waittime;
               change $T$' start time to endwait;
               call dynamic scheduling algorithm to construct a new schedule;
               if ( $T$ cannot meet its deadline)
                  return(CANNOT_MEET_DEADLINE);
               else
               {
                  create a dummy thread corresponding to the
                  first slot assigned to $T$ in the new schedule;
                  release the processor and put $T$ into rtlock' waiting queue;

                  /* code after waken up, a rough outline */
                  acquire rtlock;
                  r = func(arg);
                  release rtlock;
                  return(SUCCEED);
               }
            }
        }
}
```

4.2 Nested Locks and Deadlocks

It is not unusual that a thread acquires a real-time lock while it is holding one or more other locks. We call such locks nested locks. The structure of the RTlock function ensures that the lock holding time intervals of two locks are either non-overlapping (i.e., not nested) or totally nested. As a result, for two nested locks, we **require that the maximum waiting time and the lock holding time of the inner lock are both included in the lock holding time of the outer lock.** With this requirement and locking structure, the algorithm presented above generalizes to nested locks. To demonstrate, consider the following example of a thread T_i requesting a real-time lock L_1. Before the thread actually acquires the lock, it is queued and scheduled somewhere with $SLOT_1$ being the first slot assigned to it (see Figure 2 (a)). Next, when applying the algorithm that calculates the waiting time for a second thread acquiring L_1 after thread T_i, it is expected that T_i releases the lock within the time interval of $SLOT_1$. However, after T_i is granted L_1 and therefore resumes its execution, it may be preempted again somewhere within $SLOT_1$ if it attempts to acquire another lock, say L_2, before releasing lock L_1. Of course, if lock L_2 is not currently held by another thread, such preemption will not occur. If we assume that the lock is currently being held, then the time interval of $SLOT_1$ no longer totally belongs to T_i. Some other thread, say T_j, may be executed while T_i is waiting for L_2 (see Figure 2 (b)). Fortunately, such a situation is transparent to the algorithm performing lock schedulability analysis; it can ignore such additional context switches because it is basing its analysis on specifications of maximum lock waiting time and lock holding time of lock L_1 that includes the waiting time and holding time of lock L_2. As a result, T_i will release L_1 within $SLOT_1$ as expected.

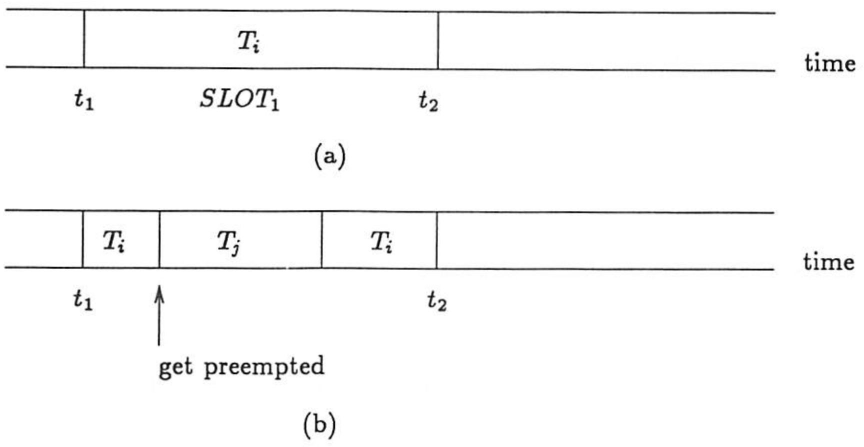


Figure 2: Nested Locks

The locking scheme described in this paper also prevents deadlocks. By performing schedulability analysis every time a thread attempts to acquire a lock, all deadlocks can be detected and avoided. We explain this with the following example. Suppose thread T_1 currently holding lock L_i requests lock L_j, while lock L_j is currently held by thread T_2. If the schedulability analysis based on t_0, the time at which T_2 will release L_j results in acceptance, then thread T_1 will proceed to wait. This implies that thread T_1 is guaranteed to acquire lock L_j at time t_0. This also implies that $t_0 < t_1$, which is the release time of L_i by thread T_1. Note that a deadlock may occur if thread T_2 waits for lock L_i before it releases lock L_j. However, this situation cannot arise because schedulability analysis must be performed before T_2 proceeds to wait. Such analysis will show that T_2 cannot acquire L_i before t_0, the time at which T_2 is supposed to release both L_i and L_j, because L_i will be released by T_1 at time t_1 and $t_1 > t_0$.

5 Performance Evaluation

Given the implementation described above, the worst case time complexity of the `RTlock` call is equal to the time for computing the waiting time for a lock, which is $O(1)$, plus the time for constructing a new schedule, which is $O(n \log n)$ where n is the number of existing threads on the corresponding processor [22]. Experimentation with the real-time threads package on the GP1000 (68020-based) BBN Butterfly shows that the minimum overhead for dynamic locking analysis is 1.18 milliseconds when the lock is not available and when there is only one thread on the processor[1]. This overhead is mostly due to our un-optimized implementation of context switching under Unix (context switching is done using the 'SETJMP' and 'LONGJMP' instructions while also saving the signal mask) and it could be easily reduced to approximately 400 microseconds total on our current machines:

1. 62.66 μsec. for atomic instructions to test the values of the two locks required for real-time lock instructions. The first lock is a spin lock used for access to the real-time lock's data structure, and the second lock is the real-time lock itself;
2. 9.54 μsec. for computing the maximum waiting time on the real-time lock;
3. 920.80 μsec. for housekeeping, which includes saving stack pointers for rollbacks in case timing constraints are violated, manipulating the UNIX signals for timing interrupts, and switching context, etc (this time could easily be reduced to approximately 150 μsec. on the GP1000 BBN Butterfly);
4. 187 μsec. for performing schedulability analysis.

Note that most of the costs listed above do not change with increases in processor workload or in the number of threads in the real-time lock's waiting queue. However, since the cost of schedulability analysis does depend on processor load, total overhead for locking analysis also increases with processor workload. This is shown by the two curves in figure 3, one depicting locking overhead including the cost of schedulability analysis, the other depicting locking overhead excluding the cost of schedulability analysis. As can be seen and as evident from the algorithm presented in the previous section, (1) locking analysis cost is independent of system workload and (2) the cost of schedulability analysis dominates, resulting in up to 12.5 milliseconds total overhead under extremely high loads (e.g., 900 threads per processor). Note that the high variations in schedulability analysis cost are due to the fact that the number of threads involved in schedule reorganization when performing locking analysis varies significantly with the current system workload.

6 Conclusion

While locking analysis cost appears discouraging to operating system implementors, it is actually acceptable for a large variety of real-time applications, especially considering that most devices for robotics (and other real-time) applications require execution rates of no more than 100 Hz even at the lowest levels of control[2]. For such applications, the highest rate threads (at the lowest levels of control) should not be written such that dynamic scheduling must be performed for resource accesses (in other words, they should be scheduled statically). However, given the timings above, even medium rate threads and their resource accesses (e.g., maximum resource access rates of 100 milliseconds) may be scheduled dynamically with no more than 10% scheduling overhead.

[1]For reference, note that a local memory reference on the GP1000 requires approximately 600 nanoseconds, a remote reference requires at least 4 microseconds, and a local procedure call may be performed in 3 microseconds.

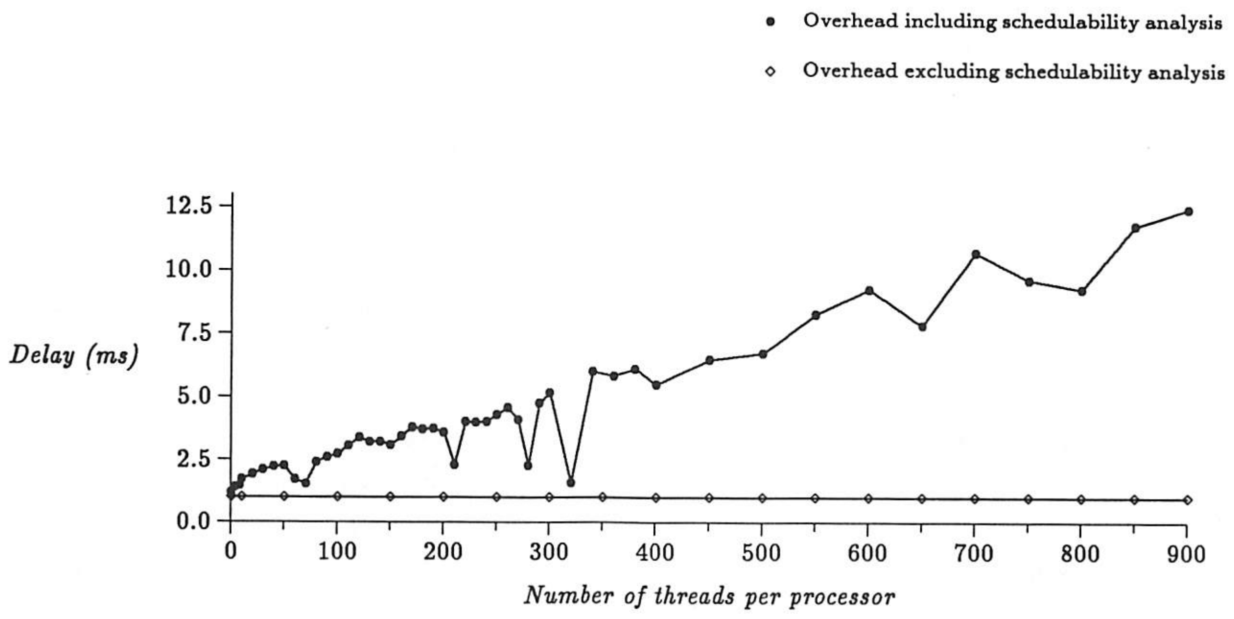


Figure 3: Real-Time Locking Analysis Overheads

This paper presents a scheme for on-line synchronization in dynamic, multiprocessor real-time applications. The scheme is shown deadlock-free (proof is given in the appendix) and predictably executable. Specifically, the algorithm that computes maximum lock waiting time for a thread attempting to acquire a real-time lock is shown efficient and easy to implement. The algorithm is embedded in a real-time multiprocessor threads package implemented on standard Unix platforms and on a 32-node GP1000 BBN Butterfly multiprocessor. Additional improvements being considered may reduce the number of rejections of lock accesses and in related work, we have generalized the uniprocessor dynamic scheduling algorithm described in [22] to deal with multiprocessor systems[2, 23]. Future work regarding threads synchronization should concern faster algorithms for on-line decision-making regarding thread scheduling. We hope to achieve improvements (1) by relaxation of timing constraints, permitting soft rather than hard deadlines, and (2) by adding some restrictions on threads' resource accesses. In addition, we are developing specialized scheduling and synchronization algorithms for specific application domains like discrete event simulation and others.

References

[1] T. Bihari, D. Pugh, T. Walliser, and E. Ribble. Timing analysis of a robot motion-planning algorithm. In *Seventh IEEE Workshop on Real-Time Operating Systems and Software, Univ. of Virginia, Charlottesville*, pages 104–107, May 1990.

[2] Ben Blake and Karsten Schwan. Experimental evaluation of a real-time scheduler for a multiprocessor system. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[3] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *The Journal of Real-Time Systems*, 2:325–346, 1990.

[4] Eric C. Cooper and Richard P. Draves. C threads. Technical report, Computer Science, Carnegie-Mellon University, CMU-CS-88-154, June 1988.

[5] Michael L. Dertouzos. Control robotics: the procedural control of physical processes. In *Proc. of the IFIP Congress*, 1974.

[6] Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing, Chicago, ILL*, pages 462–469, June 1989.

[7] Ahmed Gheith and Karsten Schwan. Chaosart: A predictable real-time kernel. In *Butterfly Users Group Meeting, BBN Advanced Computers INc., Rochester, NY*, April 1989. Talk abstracts do not appear in proceedings.

[8] Prabha Gopinath, Tom Bihari, and Karsten Schwan. Object-oriented design of real-time software. In *10th International Real-time Systems Symposium, Los Angeles*, pages 194–201. IEEE, Dec. 1989.

[9] W. A. Horn. Some simple scheduling algorithms. *Naval Res. Logist. Quart.*, 21:177–185, 1974.

[10] Kevin Jeffay. Analysis of a synchronization and scheduling discipline for real-time tasks with preemption constraints. In *Proceedings of Real-Time Systems Symposium, Santa Monica, California*, pages 295–305. IEEE, December 1989.

[11] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of Real-Time Systems Symposium, San Jose, CA*, pages 261–270. IEEE, 1987.

[12] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[13] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment.* PhD thesis, M.I.T., 1983.

[14] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[15] Richard L. Phillips. Mediaview: An editable multimedia publishing system developed with an object-oriented toolkit. In *the Summer 1991 Usenix Conference*, pages 125–136, 1991.

[16] R. Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of Real-Time Systems Symposium, Huntsville, AL*, pages 259–269. IEEE, December 1988.

[17] Yiannis Samiotakis and Karsten Schwan. A thread library for the bbn butterfly multiprocessor. Technical report, Department of Computer and Information Science, The Ohio State University, OSU-CISRC-6/88-TR19, June 1988.

[18] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5(3):189–231, Aug. 1987.

[19] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Multiprocessor real-time threads. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-91/31, July. 1991.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[21] Wei Zhao, Krithi Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, August 1987.

[22] Hongyi Zhou and Karsten Schwan. Dynamic scheduling for hard real-time systems: Toward real-time threads. In *Proceedings of Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*. IEEE, May 1991.

[23] Hongyi Zhou, Karsten Schwan, and Ian Akyildiz. Performance effects of information sharing in a distributed multi processor real-time scheduler. Technical report, College of Computing, Georgia Institute of Technology, GIT- CC-91/40, Sept. 1991.

Application Specific Coherence Control for High Performance Distributed Shared Memory [*]

R. Ananthanarayanan *Mustaque Ahamad* *Richard J. LeBlanc*

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332
Ph: (404) 853-9390
e-mail: ananth@cc.gatech.edu

Abstract

Shared Memory is an attractive alternative to message passing for structuring distributed applications. However, the performance of applications written for shared memory is poor when compared to the performance of the same applications implemented using message passing. One of the reasons for this inefficiency is the lack of use of application specific knowledge in maintaining coherence of shared memory. When a fixed number of protocols are implemented by the system, programmers are unable to match the coherence specifications to the requirements of the application.

In this paper, we consider examples of programs that use shared memory and analyze the coherence requirements that they actually impose on shared memory. We present a cost analysis of these applications using existing distributed shared memory protocols. Our initial thrust in solving the efficiency problem is towards developing primitives that can be supported at the operating system level to maintain coherence of shared memory. These primitives can be used to exploit application semantics and can be either explicitly inserted by the programmer or by transformation programs. Thus, the primitives are the mechanisms that allow users to implement the policies that correspond to the coherence protocols. Initial results from in implementing these primitives are encouraging.

An alternate view of memory coherence is to consider how the events in the memory system relate to synchronization. We outline how these events can be used to obtain the performance benefits without complicating the programming of distributed applications.

1 Introduction

A loosely coupled distributed system consists of a set of processors with private memory that communicate through a network. Two well-known techniques for inter-process communication in such a system are message passing and distributed shared memory. The model provided in a message-passing system consists of processes with private address spaces communicating through messages. A distributed application can be programmed using a set of processes which synchronize and exchange state information using messages. Although

[*]This work was supported in part by NSF contracts CCR-91-06627 and CCR-86-19886.

processors do not share physical memory, it is possible to implement a shared memory abstraction. In such a shared memory system, the address space of each process consists of both a private space and space that is shared with the address spaces of one or more other processes. In this case, the processes of a distributed application have a uniform way of accessing state information, whether local or belonging to another process (which may be executing on a different processor): by examining memory.

Remote Procedure Calls(RPC) is another way of structuring distributed applications. RPC's hide message-passing by providing a procedural abstraction in place of sending messages. That is, instead of a process sending a message to another, which subsequently processes the message by doing some computation, the computation is encapsulated in a procedure inside the second process, and the first process directly calls (RPC) this procedure. RPC is more structured than sending messages but different mechanisms are used for operating on shared state: directly examining memory for local state and using procedural calls for remote state.

A shared memory abstraction in a distributed system, called *distributed shared memory* (DSM) [LH89], is desirable not only because it provides a uniform mechanism for accessing both local and remote state but also because it offers several other benefits. For example, if a server that implements RPCs is executed by several threads, DSM allows these threads to be scheduled at different processors to speed up execution of pending calls. In a message-passing or RPC system without DSM, the instances of the server at different processors will have to be programmed explicitly. The claim that DSM simplifies distributed programming is supported by the experiences of a number of researchers [ACD+90, BCZ90, Che86].

The DSM abstraction is implemented by treating the private memory of the processors as caches of the globally shared memory and enforcing coherence of these caches through the use of appropriate protocols whenever processes read from or write to the shared memory. Several common DSM protocols are adapted from multiprocessors and do not perform well due to the lack of capabilities in distributed systems that are easily and efficiently implemented in hardware in multiprocessor systems. Thus, the cost of a simpler programming model in DSM systems is the overhead of maintaining the coherence of the shared memory and resulting loss in performance. Recently, two general approaches have been proposed for reducing the performance degradation in DSM systems:

- Restrictions on the program behavior for its correct execution on a particular memory model. This approach is best exemplified by the *Release Consistency* model in which memory accesses are classified in two categories: *synchronization* and *data* accesses [GLL+90]. This approach permits a more efficient implementation of data accesses but programs only execute correctly if they follow the required synchronization model.

- Additional information provided about the use of shared memory. For example, in the Munin system [CBZ91], data items are classified by the programmer according to how they are shared, and rather than using a general protocol, and the coherence of each data type is implemented by a specialized protocol.

Although the above approaches lead to much more efficient DSM coherence protocols, performance of applications that use DSM is still poor when compared to similar applications written for message based systems. This can be easily explained by observing that a programmer explicitly deals with data transfer in message systems. For example, a set of clients can request data from a server and read it. The server generally does not notify the clients when the data value changes because they must explicitly request the new data from the server when necessary. On the other hand, a DSM protocol will try to reflect the change in the data value at all nodes where copies of the data exist. This results in invalidation or update messages which will be wasteful if the clients do not need to read the data again.

Our goal is to provide a DSM system that allows users to develop applications that exhibit performance comparable to an implementation of the application on a message passing system. Instead of identifying a single program model or a fixed set of data types and coherence protocols to reduce the cost of maintaining data coherence, our work focuses on operating system level primitives that can be used by programmers to manage the coherence of shared data. The following are the main issues that addressed in this paper:

- We analyze the performance problems of DSM systems by analyzing sample applications which help us to identify the different scenarios under which the existing protocols do not perform well (section 3).
- We identify operating system primitives which can be used to implement shared memory coherence without loss of performance, by customizing the coherence of memory according to the needs of the application. The primitives are to be directly used by programmers or automatically inserted into programs that assume a particular program and memory model (section 4).
- An implementation of the above primitives is described and performance figures are presented (section 5).
- We discuss how the task of programming can be simplified by inserting our primitives in in the code when the programming model is restricted (section 6).

Before presenting the main issues of the paper, we revisit some of the work in the area of distributed shared memories and memory consistency in the next section.

2 Related Work

Several researchers have proposed and implemented protocols for DSM systems. In his pioneering work in this area, Kai Li implemented and evaluated several variants of an invalidation-style protocol for multiprocessor cache consistency in a distributed environment. In the Clouds system [DCM+90], since concurrently executing threads in the same object may be at different nodes of a loosely coupled system, an underlying DSM facility maintains the coherence of data shared by such threads. The Clouds protocol exploits synchronization information in maintaining memory coherence by combining data transfer with synchronization requests [RAK89]. External pagers in Mach can also be used to share data across nodes [RTY+87]. The Munin system implements DSM on top of the V distributed system [CBZ91]. As described earlier, Munin exploits information about how a particular data item is shared between nodes to reduce the cost of maintaining its coherence. Other DSM systems are reported in [NL91].

Large scale multiprocessor systems also need to deal with problems similar to the ones that arise in DSM systems. In recent research, systems have been proposed that exploit synchronization information to reduce the cost of memory coherence maintenance [DSB88, GLL+90, AH90, LR91]. For example, as mentioned earlier, the DASH project only guarantees strict coherence for synchronization variables and writes to other variables can be propagated without blocking the processor that writes these variables (the propagation of the new values must be done by the time a certain type of operation is executed by the processor on a synchronization variable). These systems can execute programs written for coherent memory correctly as long as the programs follow the appropriate synchronization model. Thus, programmers do not see the weakened memory model. Other weak memory models have been described in [AHJ91, LS88].

Since we are dealing with a loosely coupled environment where nodes do not share physical memory, we are interested in protocols that can be used in an operating system or by the runtime support system of a programming language. These protocols are software-based and certain functionality easily realized in hardware-based solutions cannot be considered. For instance, writes to shared memory cannot be tracked as in a write-through cache coherence protocol. The emphasis of our work is not on developing a new DSM protocol or to define a new memory model but to identify, implement and evaluate operating system primitives that can be used to implement a variety of data coherence protocols. These primitives will allow users to tailor the coherence requirements of shared data. Thus, our work is unique in that it segregates coherence mechanisms and allows policies to be specified at the user level. Furthermore, when we restrict the program model to automatically transform shared memory programs to ones that use the primitives proposed by us, our goal is to develop efficient software-based implementations of synchronization mechanisms so that they can be used to maintain the coherence of other data efficiently. This differs from the approach taken by the hardware designers, where the synchronization operations are also of the type read and write but their execution ensures that the variables on which they operate are kept consistent.

3 Coherence Mismatch

In this section, we argue that for any given DSM protocol, there exist application programs with coherence requirements that do not match the way the DSM protocol implements the data coherence. This mismatch results in the loss of performance for the application which can be measured in a distributed system by the number of messages exchanged by the protocol to maintain coherence. We consider two common protocols: write-update and write-invalidate. The memory coherence provided by the write-update and write-invalidate protocols, however, is the same — sequential consistency [Lam79]. To illustrate the mismatch problem, we consider several examples. First, we present an implementation of a linear solver, a problem which often arises in a variety of numerical applications.

3.1 Linear Solver

A linear solver computes a solution to a set of linear equations of the form $Ax = b$ where A is an $n \times n$ matrix of real numbers, b is a vector of n real numbers, and x is a vector of size n of the unknowns to be computed. We consider a parallel version of the Gauss-Siedel algorithm for solving the linear equations. In this algorithm, new values of the elements of x are computed iteratively until the solution converges. Let us denote by x_i^t the value of x_i

computed in the t^{th} iteration. Also, let $a_{i,j}$ be the element in row i and column j of matrix A, b_i be the i^{th} element of vector b. The values of x in the $t+1^{th}$ iteration are computed using the following relationship:

$$x_i^{t+1} = \frac{b_i - \sum_{k=1}^{i-1} a_{i,k}x_k^t - \sum_{k=i+1}^{n} a_{i,k}x_k^t}{a_{i,i}} \tag{1}$$

The computation terminates when the following convergence condition is satisfied.

$$\sum_{i=1}^{n} abs(x_i^{t+1} - x_i^t) < error\_tolerance$$

The pseudo-code that implements this algorithm is shown in figure 1. The workers are the processes that compute the new values for each element of x and they synchronize via the coordinator process. Since we use the synchronous iterative algorithm, the coordinator ensures that a new iteration is started by a worker only after all the values in the previous iteration have been computed. Although each worker should compute a number of elements, to simplify the program, we assume that each worker computes a single element (worker i computes x_i). `wait` B is the notation for `while not` B where B is a boolean condition.

```
global
  $a[n,n]$        : matrix of real
  $b[n]$          : vector of real
  $x[n]$          : vector of real each initially 0
  $complete[n]$   : vector of boolean each initially $F$
  $changed[n]$    : vector of boolean each initially $F$
  $done$          : boolean initially $F$

  COORDINATOR

local
  $converged()$: returns boolean

while ($\neg done$)
  wait ($\forall i\ complete_i$)
  if ($converged()$)
    $done := T$
  $\forall i\ complete_i := F$
  wait ($\forall i\ changed_i$)
  $\forall i\ changed_i := F$

  WORKER$_i$

  $t_i$: real initially 0

while ($\neg done$)
  $t_i := \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j - \sum_{j=i+1}^{n} a_{i,j}\, x_j\right) / a_{i,i}$
  $complete_i := T$
  wait ($\neg complete_i$)
  $x_i := t_i$
  $changed_i := T$
  wait ($\neg changed_i$)
```

Figure 1: Linear Solver : synchronous iterative solution

Before we analyze the algorithm, we define some necessary terminology. A page is the smallest unit of memory on which coherence is enforced. Each page in the shared memory system is *owned* by a particular node. *Copyset* is a list of nodes having a cached copy of a page and the list is maintained by the owner node [LH89]. We assume standard implementation of the write-update and write-invalidate schemes.

The message cost of the two schemes in each iteration of the algorithm is as follows. Since the handshake bits $complete_i$ and $changed_i$ are used for synchronization, we exclude the messages generated when these variables are accessed (when each worker computes a large number of elements of x, the synchronization cost will not be significant). To simplify the analysis, we assume that each x_i is in a separate page owned by the corresponding worker. We analyze the message cost for each iteration of a worker; total message cost for an iteration of the algorithm can be obtained by multiplying the cost by the number of workers. In particular, we will consider worker i.

First, we consider the coherence cost of the invalidation scheme. In the steady state, at the beginning of an iteration, all x_j's ($j \neq i$) have to be fetched to the node where worker i runs because cached values were invalidated when these variables were written at the end of the previous iteration. This fetching involves $n-1$ request messages and $n-1$ replies. Once the coordinator indicates that it has read all the newly computed values to test for convergence, the worker i writes the newly computed value into x_i. This generates $n-1$ invalidation messages to other worker nodes. An additional invalidation message is sent to coordinator node when t_i is written. Thus, the total message cost is $3n-2$ for each iteration for worker i.

In case of the write-update protocol, at steady state, the data at a node is always consistent with the latest value. Although $2(n-1)$ messages are needed to perform the initial fetching of data to the worker nodes in the first iteration, in the steady state the only messages generated are due to the updates sent to other workers when new values of x_i are produced. Since there are $n-1$ other workers and one coordinator, each iteration of a worker costs n messages.

Clearly, write-updates are better for executing the linear solver application. However, such arguments do not universally hold true. In fact, it may not be possible to clearly state which of the protocols will perform better. To illustrate this, we present another example.

3.2 Bounded Buffer

In the bounded buffer example (figure 2), a producer executes *AddItem* to insert items into the bounded buffer. Consumers execute *DeleteItem* to take out an item. To simplify the discussion, assume that each element of the buffer (*buf*) is located in a separate page and that all the data is owned by a data server (DS), which is different from the nodes where the producers and consumers execute.

First, we consider the overhead involved in maintaining consistency of buffer elements. Initially a producer at a node, say, N_0 adds an item to the bounded buffer. This action involves a page fetch request message from N_0 to DS and the data transfer from DS to N_0. Next, a consumer runs at node N_1, which generates a page fetch message to DS. DS forwards the message to N_0, which has the latest copy. N_0 sends the page to N_1. Thus, for each execution of *DeleteItem*, a cached copy of the page containing the consumed buffer element is created. With the invalidation scheme the cached copy is invalidated as soon as the buffer element is written into by a producer on the reuse of the bounded buffer element. Counting the invalidation as part of the read, each execution of the consumer involves 4 messages (get, forward, data-transfer, invalidation).

```
Shared Object $BoundedBuffer$;
        $buf$ :  array [0..n-1] of item;
        $InMutex, OutMutex, empty, full$ :  semaphore;
        $InIndex, OutIndex$ :  integer;
Operation $Init$;
begin
        $InIndex := 0; OutIndex := 0;$
        $empty := n; full := 0;$
        $InMutex := 1; OutMutex := 1;$
end

Operation $AddItem$(in $data : item$);
begin
        P($empty$); P($InMutex$);
        $buf[InIndex] := data; InIndex := InIndex + 1 \; mod \; n;$
        V($InMutex$); V($full$);
end

Operation $DeleteItem$(out $data : item$);
begin
        P($full$); P($OutMutex$);
        $data := buf[OutIndex]; OutIndex := OutIndex + 1 \; mod \; n;$
        V($OutMutex$); V($empty$);
end
end $BoundedBuffer$;
```

Figure 2: Bounded Buffer

Using the update scheme, all cached copies have to be updated, even though only one read is ever executed on the copies before they are updated again. For example, consider the reuse of buffer elements: a producer at N_1 writes into a buffer element, say $buf[i]$, and later a consumer at N_2 reads from $buf[i]$. At this point, with the update scheme, cached copies exist at N_1 and N_2. Now, a different producer, say at N_3, tries to write into $buf[i]$ due to the reuse of buffer elements. With a simple update scheme, updates are sent to all nodes containing cached copies. The following variations are possible when N_3 writes $buf[i]$.

- Copy at N_1 is invalidated; Multiple writers cannot simultaneously hold cached copies.
- Cached copy is retained at N_1; Multiple writers simultaneously hold copies and updates are serialized using synchronization and/or through a single data server.

With the first variation, updates, in step 5, are sent to N_2 only. Updates are sent to N_1 and N_2 in the second variation. Updates to N_1 are redundant, since producers are *pure* writers. The updates sent to N_2 could be useful, if the consumer running at N_2 happens to read from $buf[i]$ (again); however, such an assumption cannot be guaranteed. Variation 2 of the update scheme has the disadvantage of not taking care of reuse of the data-item (read/write-runs) by the writer. Strictly speaking, this is not a disadvantage when the data-item is a buffer element, since producers are pure writers of the buffer. However, if the data is being read in a run, the update will be useful and necessary. For example, consider the shared variables *InIndex* and *OutIndex*. As before, we assume these variables are each in a separate page. *InIndex* is potentially cached at all nodes where the producers execute. Let us assume that there are N such nodes and that each producer is continuously active at each of these nodes; that is, once started the producer executes forever.

In the case of invalidation protocol, since all producers are writers of *InIndex*, at most one node has a cache of *InIndex*. Thus, each access to *InIndex* incurs 3 messages : the request to DS, forward from DS to current keeper (where the last access to *InIndex* was made by a producer) and data transfer from keeper to the requesting node. The keeper node invalidates its copy on forwarding the data. In case of update protocol, the cached copies are not invalidated. Hence, at a steady state, all N nodes have cached copies. Thus, every write to *InIndex* will incur N messages: update message to owner, and $N - 1$ propagates to keepers. Thus, for this example, the update protocol performs poorly.

3.3 Discussion

We can characterize the access patterns in the above two examples as follows: the linear solver is an example where for a particular data item, there is a single writer and multiple readers. The bounded buffer is an example of multiple reader-writers with respect to *InIndex* and *OutIndex* variables. With respect to the buffer elements, the access pattern is that of a single pure writer (producer) and a single reader (consumer). We conclude that in the case of a single writer and multiple readers, the update protocol will outperform the invalidation protocol as long as there is at least one read by each of the readers for every three writes by the writer.

Invalidating or updating of potentially incoherent data are two fundamental techniques that can be employed in maintaining coherence of shared distributed data. However, the environment in which it is implemented may determine the effectiveness of either of the technique. Invalidate protocols are suited to implementation in a distributed setting, where no special hardware support is available. Once the cached copy of the incoherent data is invalidated, the correct copy can be supplied when a subsequent request is generated for that data, through a page fault, for example. However, update protocols are difficult to implement without hardware support. First, it is cumbersome to implement tracking of every write, since each update to cached data should be sent to nodes in the copyset. Second, updates are sent to nodes which no longer use the data, but the data exists in their caches due to relaxed garbage collection of pages. For these reasons, update protocols have to be assisted by program directives, so that the system will know when and where to send the update messages.

In spite of the drawbacks, update protocols are promising in one respect: the number of messages involved. In a way, updates directly reflect the message passing nature of the underlying system. Updates can be thought of as sending a message containing the state that the application wishes to share among the different parts of the program. Hence, when used carefully, we can expect the update protocol to perform as well as any message passing implementation of an application. In contrast, the invalidate protocol involves two 'extra' messages, to achieve the same effect: the invalidate message to a node caching a

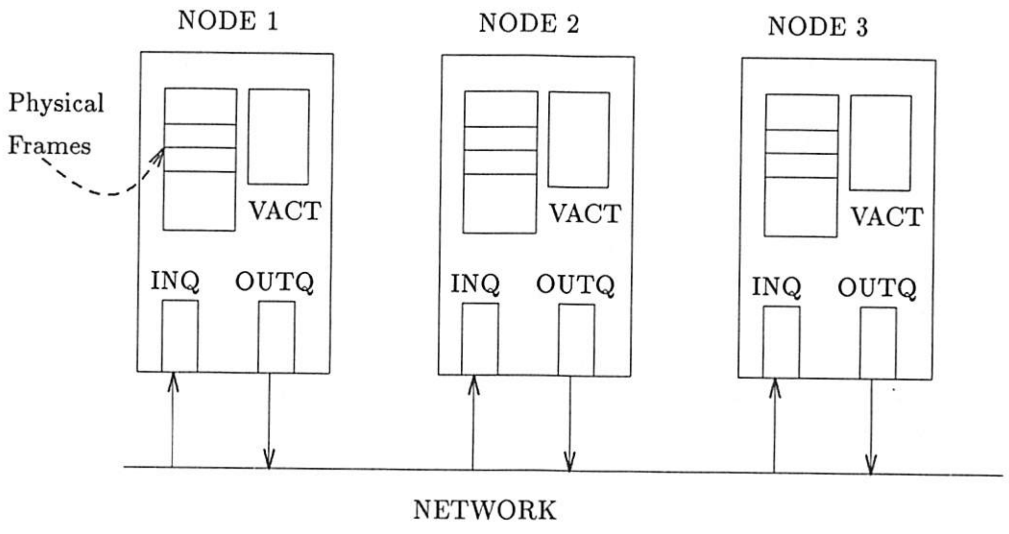


Figure 3: Model of Distributed System

given page and the get message for the same page on subsequent access by the node. This is a fundamental problem with invalidate protocols and cannot be solved with any amount of assistance from the applications. However, we believe that the update protocol can be modified, again with assistance from program directives, such that the disadvantages cited above can be mitigated, resulting in an efficient coherence scheme that reflects the nature of the application.

The invalidation technique may still be useful under certain scenarios. In some cases, nodes to which the update is sent may not need the information at the point of the update: either the node may not need it at all or may not use it for a 'long' time. In such a situation, it is better to invalidate the copy of the page at the node so that the node is forced to explicitly request the page when it is needed. Invalidating a copy makes the reader request the data on demand, while, update pre-prepares the data for the user. Thus, invalidation is preferable when the access pattern of shared memory is not known, and dynamic coherence maintenance activity is required.

Although we used the two common protocols to argue that no single protocol can perform well for all types of applications, the same can be shown to be true when other protocols are considered. We believe that since the programmer of an application best knows how data is shared, it is necessary to provide primitives that allow programmers to customize the coherence of the shared data. In the following sections, we consider a 'first cut' attempt at specifying features which can be used to control memory coherence operations. In later sections, we develop a programming model which allows automatic initiation of coherence operations from programs which follow certain constraints.

4 Basic Distributed Computing Model

In this section, we present a simple model of a distributed system. The model is intended to be general and we characterize the memory management with a functional description and procedural interfaces. Later, in the section on implementation, we give an instance of the model.

The distributed system is composed of a number of workstations connected by a network (see figure 3). Each of the machines in the system consist of primary memory and one or more CPUs. The workstations do not share physical memory and an implementation of a DSM treats the primary memories of the workstations as caches of the global DSM address space. The primary memory is divided into a number of physical pages. The size of a physical page is determined by the Memory Management Unit (MMU). A page is also the smallest unit of memory for which protection can be specified through the use of page table entries (PTE). A PTE typically maps a virtual page into a physical page with a specific protection attribute. Common protection attributes are read-only, read-write and invalid.

The CPUs generate read and write requests to memory and for each specify the virtual address of the location that is to be read or written. In either case, the virtual address is translated into a physical page number and a byte (or word) offset within it by looking up the address in the page table. If an address cannot be translated, a *page fault* is generated.

A page fault is serviced by filling a free physical page with appropriate data and setting the PTE mapping of the faulting virtual page to map to the physical page with suitable protection attributes. The 'appropriate data' is determined by the software tables maintained by the operating system. Such a table essentially specifies the characteristics of the virtual address space. Let us call this the Virtual Address Characteristic Table (VACT). Some of typical characteristics are zero-filled data, data backed by the swapping system and data to be demand paged from the disk. The VACT is a structure that could be used to characterize a virtual page as belonging to the distributed shared memory system. Note that due to characteristics of the MMU dictating that the smallest unit of protection be a physical page, the VACT cannot effectively characterize virtual address regions of less than a page size.

In general, we can classify the pointers to 'appropriate data' from the VACT as pointers to *backing stores*. A backing store provides storage for pages of memory and is said to *own* the page of memory. The backing store could use a part of the main memory itself as a store (sometimes called a *ram disk*) or could use a secondary storage device. We call the node that controls the backing store for a particular page its *owner*. A backing store should offer at least the following interface to the Virtual Memory System, particularly the page fault handler:

```
get(token, physical page)
token = unget(physical page)
```

`token` is an identifier used to name the data that is to be requested. The `get` routine fetches the data identified by the token from the owner and copies it into the physical page provided. The `unget` routine writes out the data in the physical page to the owner (when the page is dirty), and returns a token that could later be used to get the data again. Once a `unget` has been completed, the physical page could be used for other purposes. `get` and `unget` are generic routines. The DSM system, in particular, should offer such an interface.

The DSM system should also handle problems that arise due to sharing. In doing so, messages are exchanged between nodes. Outgoing messages from a node are queued in a *outq* and incoming messages are queued in an *inq*. In the next section, we present a set of primitives that can be used to maintain coherence of pages held in the caches of different nodes of a system.

4.1 Primitives

We propose six primitives that can be used to program the DSM system. These are as follows :

get(page, mode) : gets the page (the page is a pointer in the DSM shared space) in the specified mode from the owner. Mode can be read-only or read-write. The owner adds the requesting node to the list of nodes (copy-set) having a cached copy. Notice that the page, once received, will be stored in a page of the physical memory at the requesting node and the appropriate PTE will reflect this.

unget(page) : a message containing a notification is sent to the owner and the cached copy is invalidated. If page was brought in read-write mode and if the page was modified (dirtied), the notification also includes the modified page. The owner node removes the requesting node from the copy-set.

invalidate(attributes, page, node-set) : sends invalidate messages to specified nodes; attributes and node-set are discussed below.

update(attributes, page, node-set) : updates the cached copies of the page in the set of nodes specified using the contents of the locally cached copy.

getcopyset(page) returns node-set : returns the set of nodes that have a cached copy of the specified page; involves a request/reply message exchange with the owner.

flush(page) : flushes the outq of messages belonging to the page. Such messages could be updates, invalidates, etc. and are processed in the order in which they were queued. As a special case, page could be ALL, in which case all the messages in the outq are processed.

node-set is a list of nodes to which the invalidation or the update message is to be sent. It can be a subset of the nodes at which the cached copy of the page is kept. The actual nodes in the set are determined by the set of nodes which can potentially read an invalid data copy. As a special case, node-set ALL sends invalidate or update message to all nodes having a cached copy of the page (the nodes in the copy-set, maintained by the owner).

Attributes is used to classify the invalidate or update request. One of the attributes is the time at which the request is to be carried out. *Synchronous* requests are processed immediately. *Asynchronous* requests are queued in the outq and are processed later on a flush request.

These primitives are intended to provide the basic functionality for cache management. Unless explicitly directed through these primitives, the DSM system does not perform any coherence maintenance activity. Given these primitives, it is relatively easy to obtain the performance benefits of write-update or the write-invalidate protocol. The programmer analyzes the pattern of interaction in the program, chooses the appropriate primitives and inserts them in the code. Figure 4 shows the linear solver with the primitives inserted; all calls are assumed to have the synchronous update attribute.

```
global
  (* $a$ and $b$ declared as before *)
  (* $x$, $complete$ and $changed$ declared as partitioned array's *)
```

```
COORDINATOR

local
  $converged()$: returns boolean

while ($\neg done$)
  wait ($\forall i\ complete_i$)
  if ($converged()$)
    $done := T$
    update(pageof($done$), $ALL$)
  $\forall i\ complete_i := F$
  $\forall i$ update(pageof($complete_i$), $ALL$)
  wait ($\forall i\ changed_i$)
  $\forall i\ changed_i := F$
  $\forall i$ update(pageof($changed_i$), $ALL$)
```

```
WORKER$_i$

  $t_i$: real initially 0

while ($\neg done$)
  $t_i := \left(b_i - \sum_{j=1}^{i-1} a_{i,j}\, x_j - \sum_{j=i+1}^{n} a_{i,j}\, x_j\right) \Big/ a_{i,i}$
  $complete_i := T$
  update(pageof($complete_i$), $ALL$)
  wait ($\neg complete_i$)
  $x_i := t_i$
  update(pageof($x_i$), $ALL$)
  $changed_i := T$
  update(pageof($changed_i$), $ALL$)
  wait ($\neg changed_i$)
```

Figure 4: Linear Solver : with DSM primitives inserted

It is easy to see that the number of messages for coherence maintenance is minimal. Similarly, the invalidate primitive can be used to maintain coherence of shared variables in the bounded buffer example while optimizing the performance.

The update primitive has several interesting characteristics: First, it is a modified form of the *send* primitive in a message passing model. However, the *receive* primitive of the message passing model has no analogous primitive in our model: the 'message' is received by checking for the changes in the value of the memory being updated. For example, each worker i in the linear solver waits for the value of $complete_i$ to change to false before it writes x_i. Second, some of the updates can be combined. For instance, the update of x_i and $changed_i$ can be combined, since the 'reception' of this 'message' is performed through the checking of all $changed_i$'s in the coordinator. In other words, the value of x_i is not checked by the coordinator before the value of $changed_i$ is checked and hence the update of x_i can be combined into the DSM message that updates $changed_i$.

The getcopyset primitive can be used to reduce the number of messages in certain cases. If set of the nodes which cache a particular page do not change during the execution of a program, then the copyset for that page needs to be obtained only once; the writer can then send updated value directly to these nodes, avoiding the request to be routed through the owner. The linear solver is an example of such an application. After the end of the execution of the first iteration, the set of nodes which cache copies of any page will remain fixed. The copyset can be obtained at this point and can be used until the termination of the program. In fact, the technique even works when the set of nodes which cache the page can change, if the changes occur at known points. The new value of the copyset can then be obtained from the owner at such points.

While the primitives discussed above provide the performance benefits of a message passing system, such an approach is not without drawbacks. The user has to be completely aware of the underlying DSM mechanisms and has to explicitly activate the coherence operations. Hence, with this approach, transparency is compromised for the potential gain in performance. In section 6, we consider how more transparency can be provided without loss of efficiency.

5 Implementation

The implementation environment consists of a set of Sun 3/50's and 3/60's connected through an Ethernet. These machines run the native CLOUDS operating system, directly on top of the hardware. The hardware and the virtual memory management provide the characteristics of the model described in section 4. The MMU supports pages of size 8K. The backing store abstraction is supported by CLOUDS through *Partitions* [DCM+90]. The name of a memory page (identified by `token` in the backing store interface of section 4) consists of a system-wide unique segment name and an offset. It should be noted that although the CLOUDS system, was used as a testbed, systems such as Mach, or the augmented V system [CZ83] with programmer specified paging capability could easily serve the same purpose.

The backing store is implemented as a server which uses a disk to store the pages of shared memory. A get request is initially processed by reading the appropriate page from the disk. Once this is done, the page is kept in the memory of the server, so that subsequent get requests can be processed without the overhead of disk access. The server keeps a copyset for each page, which is a list of nodes currently caching the page.

Each node, in addition, has a slave server that processes incoming update and invalidate requests. On the receipt of an update message, the following actions are taken: software tables associated with the existing page are changed so that the address translation points to the updated page and hardware contexts (essentially PTE's) are cleared of any pointers to the old page. Thus, access to the virtual address will generate a page fault and the software tables will be used to 'pick up' the new page. This technique of swapping of pointers to a page avoids copying of data, which requires 1-2 milliseconds, as compared to a page fault service of a in-memory page, which takes 0.25-0.5 milliseconds[1]. However, the technique requires the ability to extract the page of data from an incoming message, dissociate it from the message and map it into a different virtual address region. Such functionality is provided by the RA kernel [BAHKi87] on which Clouds is built. Invalidate messages are processed in a similar manner.

Performance figures for the DSM primitives are given in Table 1. The first two entries give the cost of the system underlying the DSM system. Subsequent entries tabulate the performance of the primitives discussed in section 4.1. In all of these measurements, the server owns the data pages and other nodes fetch the pages from it. The relative performance of using invalidate and update primitives is as follows. Assume that two nodes, N_1 and N_2 share a page and that node N_1 writes into the page. To make N_2's copy consistent, if

[1]The actual time depends on whether we use a 3/50 or a 3/60.

Activity being monitored (All times are in milliseconds)	Measured Total Time
Basic context switching	0.3
Page transfer at the transport layer	24.4
Initial Page fault	
Pages in memory of server	27.7
Pages to be retrieved from disk	34.5
Updating a single dirty page	
Back to server	26.0
Server and one other node	52.3
Server and two other nodes	78.3
Updating a single dirty page directly using copy set (without server as intermediary)	
Update one other node	33.8
Update two other nodes	66.3
Page fault after an invalidate	38.3
Time to invalidate	
Notifying the server (no other nodes cache page)	9.1
One node to be invalidated	18.5
Two nodes to be invalidated	28.1
Time to get copy set from server	9.2

Table 1: Timings of Primitives

the update primitive is used, then the action would take 33.8 ms[2]. On the other hand, if an invalidate and a subsequent page fault is used, then the total cost would be 56.8 (18.5 + 38.3). Thus, the update primitive, if used appropriately, would show a performance improvement of about 40% over using invalidations.

Several issues need to be considered in the implementation. Firstly, granularity of data items in a shared memory page could cause correctness problems. Consider for instance, two data items, d_1 and d_2 co-located in a page P. The page is fetched in read-write mode and cached at two nodes, A and B. d_1 is modified at A and d_2 at B, simultaneously. When the updates from each node are sent to other node, only one of the modifications will be survive, A's changes at B and vice-versa. This problem is due to false-write-sharing of the page. Thus, it is necessary to ensure that simultaneously writable data items are not co-located on the same page. The compiler, with the aid of the programmer must place the data appropriately. For example, the language used to program the applications in this paper, Distributed C++ [Ana91], supports partitioned arrays. In the linear solver example, the vector of x_i's is such an array: it is divided into contiguous parts, one per worker. Each part of the partitioned array is placed in a different page, but the access to the array elements is provided through overloading the index operator [Str86], which hides the partitioning.

Another issue is that of visibility of the changes to shared memory. Our model assumes that unless updates (or invalidates) are requested, the other nodes do not observe the changes made to shared memory at a node. It does not specify visibility of changes to

[2]We assume that copyset was obtained at a one time cost.

processes (or threads) on the same node. If the threads share the same page at the hardware level, the changes will be visible to all processes without the update request. Thus, the update primitive should be seen as a way of guaranteeing memory consistency; it cannot be assumed that changes to memory will *not* be observed by the processes if the the update is not explicitly requested.

6 Using Synchronization Information

The primitives discussed in Section 4.1, if used directly by the programmer, take away the simplicity of shared memory programming. Ideally, the programmer should provide directives at a 'higher' level which can be used by a transforming algorithm to insert the primitives that correspond to the directives. Such an approach is also followed in the design of the Munin system [CBZ91], but the directives tag data objects primarily based on the sharing pattern of the object, as opposed to using synchronization information as a basis in our design. The rest of this section outlines the feasibility of our approach.

Events in a pure shared memory system consist only of reads and writes to shared memory. Some of these reads and writes are used in synchronizing access to other parts of shared memory. For example, in the linear solver implementation (Figure 1), `wait`$(\neg complete_i)$ in the Worker is used as a *lock* primitive and the statement $complete_i := F$ in the Coordinator is used as an *unlock*. These statements synchronize access to the variable x_i.

In an *Augmented Shared Memory* (ASM) system, other events such as general synchronization operations may affect the observability of changes to shared memory. For example, creation and termination of a process can be detected by other processes. In such a case, a programmer can assume to observer changes made by the terminating process to shared memory after detection such an event. Other synchronization operations in an ASM consist of system-provided synchronization, in the form of semaphore, locks, monitors, conditional critical regions, etc. Hence forth, we assume an ASM system.

A *weakly consistent* shared memory system [DSB86] maintains coherence of memory on synchronizing accesses rather than on every access to shared memory. While our solution is based on a similar approach, we extend the weak consistency model with the following observations:

- Some synchronization variables *exactly* protect particular pages of memory. For synchronization operations on these variables, every time the lock is granted, only a fixed set of pages are accessed. An example of this type of synchronization variable is $InMutex$ (see Figure 2) semaphore; it protects access to the shared variable $InIndex$.

- The particular pages of memory made accessible by a lock may be decided at run-time. In this case the synchronization variable is said to *approximately* protect the pages of memory. Although the exact page accessed may be 'refined' by an inner lock, and only a portion of the memory may be accessed in an exclusive manner, the outer lock protects, in general, a bigger area of memory. For example, the operation `P(`$full$`)` of the bounded buffer obtains access to a 'full' buffer element, which, at the point of the `P` operation could be any of the buffer elements. Hence, the `P(`$full$`)` operation potentially protects all of the buffer elements. In general, counting semaphores provide potential access to a set of pages of memory, whereas only a subset of the pages will be actually accessed.

- Certain algorithms follow a predefined data sharing pattern. That is, an unlock makes the shared accesses visible to certain fixed number of processes, regardless of the contention for the lock variable in question. For example, in the linear solver application, once the location of the workers and the coordinator are fixed, the sharing pattern of the data is fixed, assuming absence of process migration. Many parallel programs partition the data into a number of non-overlapping parts and a 'parent' process creates 'worker' processes to operate on the data. The workers work in phases, each phase consisting of computing and producing a new value for the data partition allocated to the particular worker, possibly depending on the previously computed values of other data partitions. This class of programs is known as Single Program Multiple Data (SPMD) programs.

We next consider automatic insertion of the primitives discussed in Section 4.1, by making use of synchronization operation and the above observations. In place of directly programming with the primitives, the programmer is required to provide the following information:

- Identify variables used in synchronization, collectively termed *synv*. These constitute shared synchronization variables, and locks and semaphores of the ASM system.
- Protect access to each non-synchronizing shared variable by using exactly one *synv*, that is, bracket all accesses to the shared variable by operations equivalent to lock(*synv*) and unlock(*synv*), such that at most one process could be writing into the shared variable at any one time. While each shared variable is protected by exactly one *synv*, a *synv* may protect access to multiple shared variables.
- Provide association of shared variables and *synv*'s. This could be either on a per *synv* basis or on a particular operation on the *synv*, and classify each association as being exact or approximate.
- Optionally, identify points in the program, in the form of unlock operations, where sharing patterns get established.

The sequence of activities that take place on lock and unlock operations are as follows: An unlock operation proceeds in two phases. During the first phase, the list of processes waiting on the *synv* is determined. Access is to be given to a set of such processes. In case of a semaphore, there is exactly one process, while in case of a read-write lock, a number of readers could be granted the lock. The set of nodes at which these processes execute is then determined. At this point, consistency maintenance activities are performed, as discussed below. After consistency maintenance activities, the processes granted the lock are allowed to proceed.

Consequently, the lock operation can be thought of as proceeding in two steps. During the first step, the request is sent for obtaining the lock. This request may result in the process being blocked. In the second step, consistency maintenance activities take place, and then the process is allowed to proceed.

The exact steps of consistency maintenance depend on the type of protection that the *synv* provides. If the *synv* protects the shared variable exactly, then the pages made accessible (which will be actually accessed) are pre-determined. Consistency maintenance, then, simply consists of updating the pages containing the shared variable using the `update` primitive to the set of nodes where the processes being granted the lock execute. If the *synv* protects the shared variable approximately, the cached copy of the particular page of the shared variable is written back to the owner by using an `unget`, and other copies of the page are invalidated by using an `invalidate`. As noted before in Section 3.3, invalidate is preferred over update since the node at which the new value of the data will be used is not known at the time of the generation of the new value.

When a directive to a establish sharing pattern is encountered along with an unlock operation, the `getcopyset` primitive is used to obtain the set of nodes that share the particular page. This information is used during the second phase of the unlock operation on the subsequent encounter of the unlock.

Figure 5 shows the bounded buffer example along with directives on the use of synchronization variables. The sequence of activities that take place during the *DeleteItem* operation are similar.

```
Shared Object BoundedBuffer;
        buf :  parray [0..n-1] of item; (* partitioned array *)
        InIndex,OutIndex :  integer;  (* allocated on separate pages *)
        InMutex :  semaphore exactly protects InIndex;
        OutMutex :  semaphore exactly protects OutIndex;
        empty, full :  semaphore approximately protects buf;

(* Operation Init as before *)

Operation AddItem(in data : item);
begin
        P(empty);               (* allows access to buf *)
        P(InMutex);             (* when allowed to proceed, InIndex has latest value *)
        buf[InIndex] := data;   (* might page fault to fetch current value,
                                   due to previous invalidation *)
        InIndex := InIndex + 1 mod n;
        V(InMutex);             (* updates InIndex at the node of first blocked process *)
        V(full);                (* Invalidates particular pages of buf that were accessed *)
end
```

Figure 5: Bounded Buffer : with synchronization directives

6.1 Discussion

Using synchronization information in the above manner has several interesting characteristics. When shared variables are exactly protected, updates are used to make the memory of the processes granted a lock consistent. The associated actions are exactly the same as the activities that take place in a message passing system; messages are sent only to processes that need the information. If the cost of synchronization is included, our approach appears to perform worse than the message passing approach, since synchronization is automatically achieved on receipt of messages. However, in a message passing system the sender has to explicitly name the set of receiving processes. Explicit synchronization in the ASM system provides a level of decoupling, since the 'sender', or the process performing the unlock does not have to know the identity of the 'receivers' statically. Thus, our approach allows for dynamic data sharing patterns. However, if the sharing pattern is well known, this information can be included in the implementation of synchronization, our approach performs as well as the message passing approach. Efficient realization of synchronization is one of the goals of future research.

In previously proposed research, synchronization information has been used to realize *release consistency* [GLL+90], for example, in Munin [CBZ91]. Release consistency requires updates to be sent to all processors on an acquire (lock) operation, that is, in the terms of the authors, *globally performed*. Our goal is to identify and implement synchronization operations such that only nodes that can obtain access to the data can see the changes to it. Thus, there may be inconsistent copies of the data in the system as long as no process accesses them. Only the *buffered consistency* model [LR91] allows such copies to exist. Intuitively, our system allows the data to *flow* along the *path* of synchronization, and does not care about inconsistent copies, as long as they are is not in this path.

7 Concluding remarks

Traditional Distributed Shared Memory systems perform poorly due to their mismatch with application requirements. Recent research on the Munin system has shown that such information can be obtained from programs without unduly complicating the programming model presented by the system. Our work focused on investigating the source of the performance problems and designing primitives to realize application specific coherence control. The use of synchronization information in the programs leads us to a way to automatically use these primitives through higher level program directives. This approach is based on notions of flow of data along with transfer of control on synchronization operations, rather than on global performance of updates found in release consistency. Since, the programmer provides only synchronization information and the association between synchronization and data usage, we believe that program complexity is kept to a minimum. Factoring out the primitives gives us the option of adding transformations of higher level program directives in the future, if better algorithms to maintain coherence based on synchronization or some other directives are found.

One of the directions of future research is to realize synchronization mechanisms efficiently. Such an effort is needed as the performance of the programs will depend not only on the efficiency of coherence maintenance operations, but also that of synchronization. In this work we have assumed that pages are not falsely shared. However, data placement to avoid false sharing cannot be performed by compilers in an efficient manner and the burden is usually passed on to the programmer. Avoiding the effects of false sharing and support for fine-grained data items are, thus, other areas of our future research.

References

[ACD+90] Mustaque Ahamad, Muthusamy Chelliah, Partha Dasgupta, Richard J. LeBlanc, and Mark Pearson. Shared memory programming in a distributed system. Technical Report GIT-CC-90/63, College of Computing, Georgia Institute of Technology, November 1990.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[AHJ91] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *11th International Conference on Dist. Comput.*, May 1991.

[Ana91] R. Ananthanarayanan. CC++ Reference Manual. Technical Report GIT-CC-91-07, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1991.

[BAHKi87] J. M. Bernabéu Aubán, Phillip W. Hutto, and M. Yousef A. Khalid i. The Architecture of the Ra Kernel. Technical Report GIT-ICS-87/35, Georgia Institute of Technology, College of Computing, Atlanta, GA, 1987.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–177, March 1990.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Che86] D. R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In *Proc. of the 6th Int'l Conf. on Distr. Computing Sys.*, pages 190–197, May 1986.

[CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed v kernel and its performance on diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[DCM+90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46, Winter 1990.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[DSB88] Michel Dubois, Christoph Scheurich, and Faye Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–22, February 1988.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LR91] J. Lee and U. Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, 1991.

[LS88] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, pages 52–60, August 1991.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 18th International Conference on Parallel Processing*, pages 160–169, August 1989.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectu ral Support for Programming Languages and Operating Systems*, pages 31–39, 0ct 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, 1986.

A Distributed Consistency Server for the CHORUS system

Vadim Abrosimov

Chorus systèmes

Francois Armand

Chorus systèmes

Maria Inés Ortega

LITP Institut Blaise Pascal - Chorus systèmes

Chorus systemes
6, avenue Gustave Eiffel, F–78182, Saint-Quentin-en-Yvelines (France)
Tel: +33 1 30 64 82 56, Fax: +33 1 30 57 00 66, E-mail: ines@chorus.fr

1. Introduction

This paper describes how distributed shared memory is being implemented on the system, by a set of servers running outside the CHORUS Nucleus itself. These servers cooperate to implement, in a decentralized fashion, distributed consistency of data between multiple sites. The algorithms used for that purpose derive from those described in[Li89a] . This service will be used to provide Distributed Shared Memory in the CHORUS/MiX V.4 subsystem running on top of the CHORUS Nucleus. The CHORUS/MiX V.4 subsystem is compatible with UNIX SVR4 and is designed to provide a Single System Image on multicomputer architectures.

The next sections will briefly describe the CHORUS system and the CHORUS/MiX V.4 distributed system, and will detail some of the reasons why Distributed Shared Memory is needed when building up a Single System Image.

We will then spend some time to summarize similar mechanisms designed and implemented in related projects. After having summarized the CHORUS Virtual Memory interface and the main goals of the design of our servers, we will describe their design and some implementation details. We will conclude with some lessons learned and outline future developments of these servers.

® CHORUS is a registered trademark of Chorus systèmes.

® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

2. CHORUS/MiX

2.1 CHORUS Architecture

A CHORUS System is composed of a small-sized **Nucleus** and of possibly several **System Servers** that cooperate in the context of **subsystems** to provide a coherent set of services and a user interface. A detailed description of the CHORUS system can be found in[Rozi88a] . Among the other systems that have adopted similar architectures one will find: Mach [Acce86a] ,V-system [Cher88a] and Amoeba [Mull87a] are some examples.

2.2 CHORUS Nucleus Abstractions

The **actor** defines an address space that can be either in user space or in supervisor space. In the latter case, the actor has access to the privileged execution mode of the hardware. User actors have a protected address space. One or more **threads** (*light weight processes*) can run simultaneously within an actor. They can communicate using the memory of the actor if they run in the same actor.

Otherwise, they can communicate through the CHORUS IPC that enables them to exchange **messages** through **ports** designated by global unique identifiers (or UI). A message is composed of a (optional) **body** and an (optional) **annex**. Ports may be dynamically inserted into or removed from **port groups**. The CHORUS IPC mechanism allows a message to be sent to all ports of the group (broadcast mode) or to only one port in the group (functional addressing mode).

2.3 The CHORUS/MiX V.4 subsystem

MiX V.4 is a CHORUS subsystem providing a UNIX interface that is compatible with UNIX SVR4. It is both BCS and ABI compliant on AT/386 machines. It is composed of a set of cooperating servers that run on top of the CHORUS nucleus and which communicate only by means of the CHORUS IPC. The following servers are the more important:

- The **Process Manager** (PM) provides the UNIX interface to processes. It implements services for process management such as the creation and destruction of processes or the sending of signals. It manages the system context of each process that runs on its site. When the PM is not able to serve a UNIX system call by itself, it calls other servers, as appropriate, using CHORUS IPC.
- The **File Manager** (FM) also refered to as the Object Manager (OM) performs file management services.
- The **Streams Manager** (StM) manages all stream files such aspipes, network access, tty's, named pipes.

CHORUS/MiX V.4 has been designed to be distributed over a set of sites. The ultimate goal of this design is to provide Single System Image semantics, masking multicomputer topologies to user process. Among the features that must be provided to achieve such a goal, one will find the following: single file name space, transparent access to any file from any node of the multicomputer, unique process identifiers name space, remote execution and process migration capabilities...

Some of these features have already been developed and proven within the previous version of the MiX subsystem (MiX V.3.2) which is compatible with UNIX SVR3.2 systems. The Locus[Pope85a] system although not based on a microkernel has also demonstrated such capabilities.

3. Needs for Distributed Shared Memory

Among the reasons to provide distributed shared memory in a UNIX SVR4 compatible distributed system, one may list the following: processes running on different sites may communicate through System V IPC shared memory mechanism, regular files being accessed concurrently from different sites need to be maintained consistent.

Unix systems permit users to map files into their process address spaces. This mapping of files can be shared between several processes. When a file is mapped it can either be read or written by simply reading or writing an address location within the process address space corresponding to the offset of the byte in the file. If the mapping is shared between two processes, every modification made by any of the two processes is immediately visible to the other one. This behavior must be maintained within the distributed system even if the two processes are running on different nodes.
In a distributed system, in order to achieve good performance when accessing remote files, it is desirable to cache, on the client side, parts of the files being accessed. This raises the problem of maintaining coherence between all these client caches. In CHORUS/MiX this problem is identical to the "mapped file" issue.

In addition, shared memory as opposed to IPC mechanisms provides a simpler abstraction to the application programmer to enable multiple processes to communicate. Thus it seems a quite desirable feature although not without potential pitfalls when misused.

4. Related Works

There are numerous works and projects which deal with distributed shared memory consistency.There is no room here to describe all of them. Generic surveys of literature dealing with this topic may be found in[Hell90a] ,[Nitz91a] ,[Stum90a] and[Tam90a] . We will focus on the work conducted in IVY[Li89a] and Leases[Gray89a] .

4.1 IVY

IVY is a study of the memory coherence problem in designing and implementing a shared virtual memory on loosely coupled multiprocessors. Shared data is paged between processors.

The protocols assume that every page is owned by a site. The ownership can be fixed or dynamic. In the fixed approach a page is always owned by the same processor. In the dynamic case the owner of a page can change but there is always only one owner at any given time, namely the last site that has had write access to the page. Page owner information is managed according to one of the following strategies: *centralised* where only one site knows all page-site mapping and, *distributed* where several sites cooperate to manage the page ownership.

The distributed policy although more complex to implement provides better throughput. The work descibed in this paper derives from this mechanism, but provides some extensions which will be described later.

IVY's solutions operate only with access on one page a time. If the Chorus mapper wants to serve many Unix servers it must make sure that the access to the object's fragments is atomic. Moreover, we support different page sizes; the granularity is not fixed. In a CHORUS system, several algorithms implementing different coherence semantics may coexist: coherence algorithms are implemented by an independant actor outside of the nucleus.

4.2 Leases

Leases proposes an efficient fault-tolerant mechanism for distributed virtual memory consistency that handles host and communication failures using physical clocks. A *lease* is a contract that gives its holder specified rights over property for a finite period of time. In the context of caching, a lease grants to its holder control over writes to the covered datum during the term of the lease. The server must obtain the approval of the leaseholder before the datum may be written.

One of the problems of this algorithm is to determine the duration of the lease. It is based on a trade-off between minimizing lease extension overhead versus minimizing false sharing. Short-term leases have a number of significant advantages over longer leases, including lower write delays resulting from client crashes, lower recovery delay from server crashes and reduced false sharing.

We think that this work would be interesting and could be incorporated in the future with our work because it provides a mechanism to avoid thrashing and fault-tolerant problems.

4.3 Miscellaneous

Mirage[Flei89a] provides a consistent distributed shared memory using infinite-term leases. Mirage's property permits the readers or the (single) writer of a page uninterrupted access to the page for a fixed period of time, regardless of other processes requesting it.

Munin[Benn90a] is a DSM system proposed and currently being developed at Rice University. Instead of a single memory coherence mechanism for all shared memory, Munin employs several different mechanisms, each appropiate for a different class of shared data object.

Another direction is a DSM accomodating heterogeneity. This is a difficult problem, because at the page level, byte and words are the primitives, not types data objects. Arcade's support of heterogeinity has led naturally to sophisticated and powerful kernel-level support for distributed shared memory using a langage approach[Cohn91a] . This problem has been also dealt with in[Zhou90a] and[Stum90a] .

5. CHORUS Virtual Memory

The CHORUS Virtual Memory (VM) guarantees local memory coherence and offers tools to build distributed memory coherence[Abro89a] . In this section we only present the VM interface that is used to build external mappers.

5.1 Basic Abstractions

A **segment** is a collection of data with an associated name (eg: a file). This name is a **capability** (segcap) exported by external servers called **mappers** (eg: the MiX File Manager) and is built from a server's port Unique Identifier and a key that is meaningful only within that server. These servers manage the implementation of the segments, as well as their protection and designation. A segment can be either expliclity read and written through the sgRead/sgWrite CHORUS system calls, or mapped in an actor's address space.

The nucleus encapsulates the physical memory, holding portions of the segment in a per segment **local cache** object (see figure 1). A **local cache** object is designated by its capability; the server for local caches is the nucleus. In addition, each local cache contains a log of all pages which have changed relative to the segment. With each page in the cache, there is associated an **access right** that describes the possible operations (e.g read/write) allowed at a given time.

On a site, the same cache object is used for both the mapped and the explicit segment access, thus insuring that any modification done through one interface will be immediately visible through the other. In other words, the unicity of the cache object avoids any double caching issue for a segment. When multiple sites use the same segment, the corresponding cache objects (one per site) will be named with different **local cache capability** (lccap), thus enabling a mapper to distinguish between these different local caches, and to implement a distributed consistency maintenance protocol.

Page faults generated by reading or writing the memory associated with a mapped object will, in turn, produce requests to the **mapper** for the corresponding access right and data from the object. When the nucleus wishes to free modified pages, it sends requests to the mapper to write back the modified data.

A **mapper** exports a simple segment access interface (described in the next section) to the nucleus. Conversely, the VM exports an interface allowing a mapper to control the state of the local cache associated with a segment. Both interfaces rely on the CHORUS IPC mechanism.

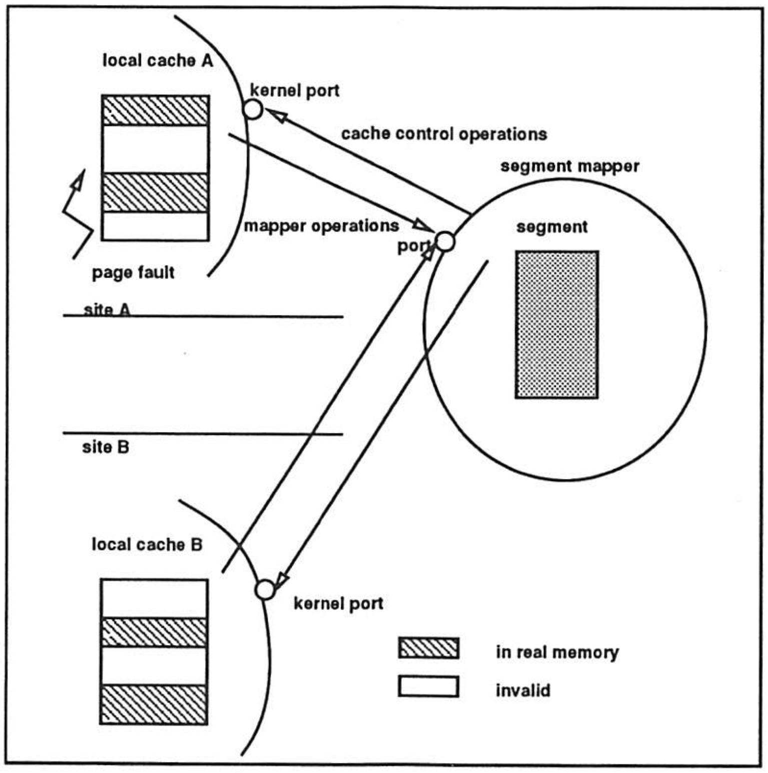


Figure 1. – Local Caches

5.2 The Segment Request Interface

The segment request interface provides a mechanism by which a nucleus can demand and return pages of a segment to a mapper. It also provides a means for determining access rights to parts of a segment, without shipping the associated data, and to page out parts of a local cache. The requests which make up this interface are:

- **mpGetAccess** (*segmentCapability, size, offset, requiredAccess:R/W*)
  The **mpGetAccess** request permits a nucleus to request read/write access to a fragment of a segment. If a legitimate access is requested then the mapper will return a success indication.
- **mpPullIn** (*segmentCapability, offset, size, accessRequired:R/W*)
  This request is used by the nucleus for demanding read/write access and the data of the segment to the mapper. If the mapper indicates success then it must supply some data together with the reply. The data returned may already be dirty (not yet saved on backing store). If the mapper indicates failure then the local cache will remain unchanged, and the page fault will not be satisfied. An *mpPullIn* request carries an implicit *mpGetAccess* request for the fragment.
- **mpPushOut** (*segmentCapability, offset, size*) ;
  This request causes data to be updated from a local cache to a mapper. Usually dirty pages are pushed out, but clean pages may also be pushed out upon explicit request of the mapper (see lcFlush below). The mapper can write the data to the segment (eg: on secondary storage) or send the data to another site which has made an *mpPullIn* request.
- **mpCreate** ()
  This request is the means by which the nucleus creates a segment for an internally created local cache (e.g. Swap). In case of success it returns the capability for the segment being created.
- **mpDestroy** (*segmentCapability*)
  This request permits the nucleus to end the association between a local cache and its segment which was created by an *mpCreate* call.

5.3 The Local Cache Request Interface

The local cache request interface provides the means for mappers to maintain control over the data they supply to local caches. We present the **lcFlush** operation which disposes of data in a local cache and **lcSetRights** which changes the access rights associated with a page.

5.3.1 lcFlush

lcFlush is called by the mapper to invalidate, to change the access rights of, or read a local cache. It takes as arguments an offset, a size, a local cache capability, and a flag. The flag indicates the mode of flush and may be: invalidation mode, invalidation mode and read-cache or change of access from write to read and read-cache. An *lcFlush* operation performed against a local cache may result in one or several *mpPushOut* requests from the CHORUS VM to the mapper. Flushing a local cache is a synchronous operation, and thus will return to the caller when the corresponding *mpPushOut* operations will have completed.

Let us focus on six relevant situations which can occur when a *lcFlush* is called:

1. The flag is *"invalidate"* and the page has not been modified (figure 2.1): in this case the page is nvalidated and the operation returns.
2. The flag is *"invalidate"*, the access rights granted to the local cache were "read", but this page has been modified and the modifications are not yet written-back (figure 2.2). This case is possible if an access rights change from write to read has previously occurred. The page is then invalidated and a *mpPushOut()* to the mapper is performed.

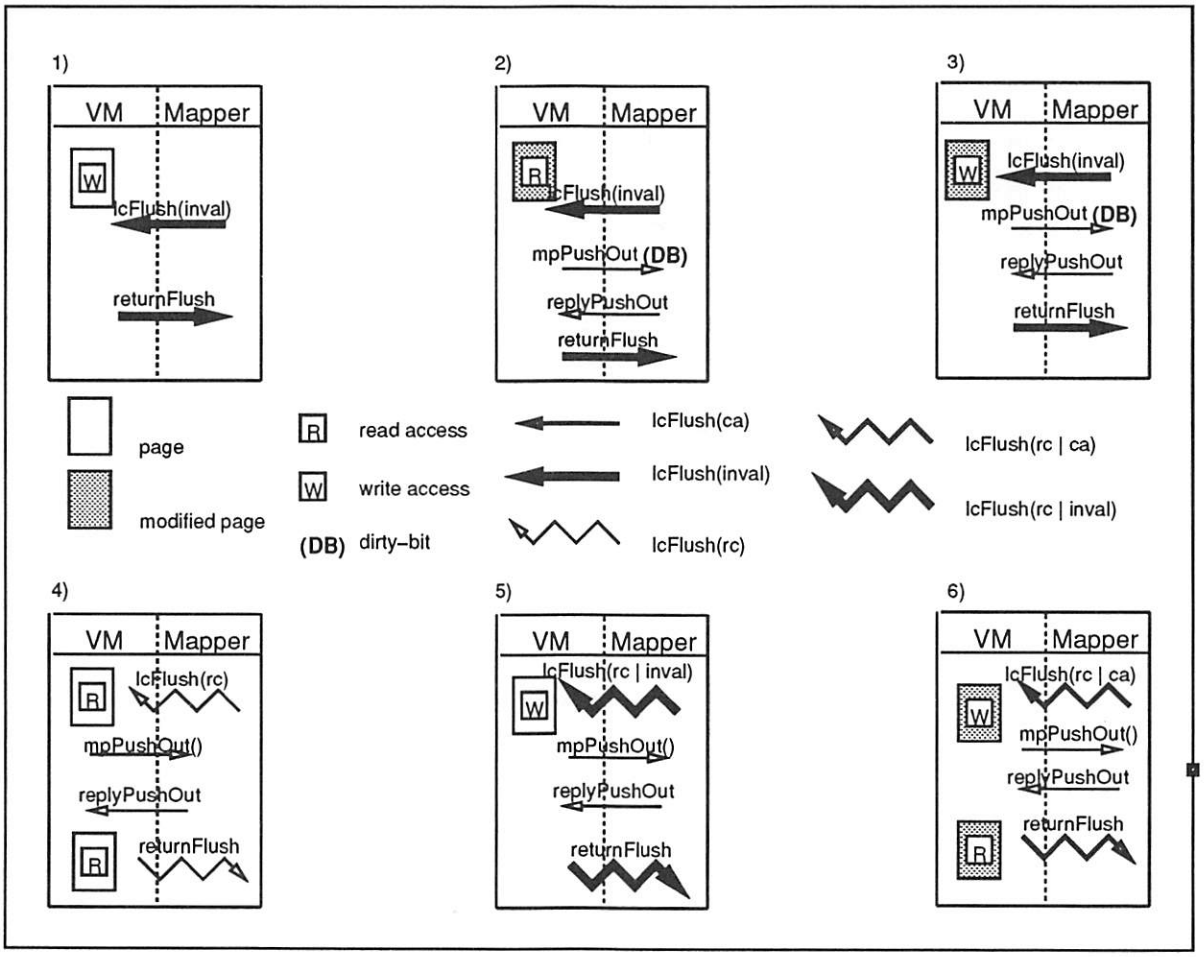


Figure 2. – Flush control

3. The flag is *"invalidate"* (figure 2.3), the access rights granted to the local cache were "write", and the page has been modified: the site loses all rights on the page and the page is pushed out to the mapper. In fact, this case is identical to the previous one.
4. The flag is *"read-cache"* and the page has not been modified: an *mpPushOut* to the mapper is peformed (figure 2.4), but the page remains in the site with the same access rights.
5. The flag is *"read-cache / invalidate"*: the page is invalidated and an *mpPushOut* is performed whether the page has been modified or not (figure 2.5).
6. The flag is *"read-cache / change-access"* and the page has been modified: the access rights change from write to read and an *mpPushOut()* occurs (figure 2.6).

5.3.2 lcSetRights

The **lcSetRights()** changes the access rights associated with the page from write to read or invalidates the data (recalls all access rights). It receives the offset and size of the fragment being involved, the capability of the segment and an optional invalidate flag.

In the next paragraph we present, two examples of the use of this function made by the mapper.

Figure 3 shows a page with write access right:

1. In the first case (figure 3.1), the page has not been modified so the access right is changed from write to read and the operation returns.
2. In the second example (figure 3.2), the page has been modified, the access right is also modified but the page remains modified with read access rights.

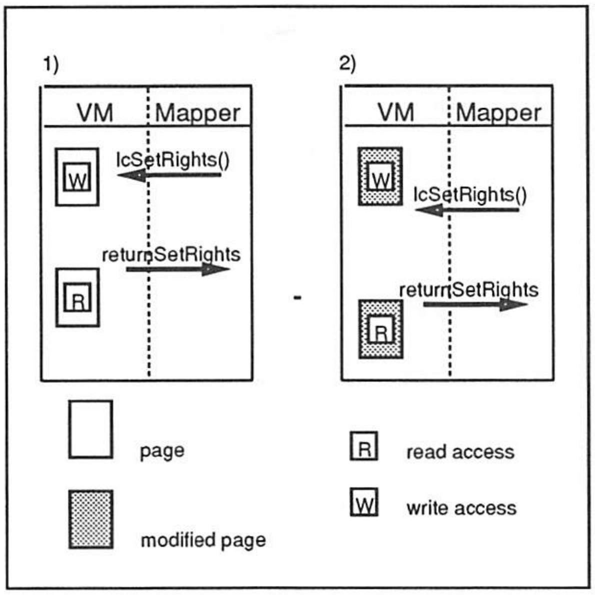


Figure 3. – Change access

5.3.3 Data Transfer Mechanism

External Mappers that provide only consistency mechanisms but no storage for segments do not need to access the data of the segment. They just need to move these data around ("receive" and "forward"). The CHORUS segment interface allows one to move data by using a "Data Descriptor" rather than by giving the address and size of the data to be moved. Thus, one can avoid mapping or copying data in a Mapper if it is not needed.

6. The Distributed Coherency Server

6.1 History and Main Goals

We have already developed a Coherency Server that implements the "centralized" algorithm as described in IVY[Orte91a] . In such a case, when a site A wants to get a page of a segment, it has to send a *mpPullIn* request to the "centralized" mapper. Let us suppose that the requested page was already loaded with "write" access granted on site B. The mapper will then have to reclaim the page from site B (using *lcFlush*) before being able to send this page back to site A.
Such a scheme although simple has the drawback to move the page twice (from site B to the mapper, and from the mapper to site A). This consumes network bandwidth and makes the mapper a potential bottleneck. Experiments done in IVY have also shown that this algorithm is

not the most efficient. Thus, we wanted to implement the "dynamic decentralized" mechanisms as described in IVY.

When designing this service we had several goals in mind:

- Independent Server
  In the same way, mappers are not part of the CHORUS Nucleus, so that system builders may implement the coherency policy they need, we wanted to have consistency be implemented outside any CHORUS/MiX File Manager. This is not only helpful to start coding and debugging, but also provide an easy way to provide distributed consistency for any file system server that exists or will exist. Such an independent server may also be used in any other subsystem that requires the same kind of consistency (eg: Object Oriented subsystems). Due to the encapsulation of the consistency policy, it is also easier to make it evolve without impacting on either the CHORUS VM or the MiX File Manager.

- Support of Unix semantics
  We had some additional requirements: the mapper should at least provide the basic mechanisms to fully support transparent UNIX semantics in a distributed environment. Mainly, one must guarantee that *read*(2) and *write*(2) operations are processed serially. In other words, a read operation occurring concurrently with a write operation on overlapping area of a file, may return data as they were before the write or as they are after the write complete. This must be guaranteed even though the read and write operations are done on large fragments of the file (i.e.: larger than a page). This is an extension to the service provided in IVY which deals only with page access concurrency.

- Heterogeneous Granularity
  In a distributed environment one may have machines running with various virtual page size (whether they have or not the same instruction sets is another issue): Sun3 machines used 8Kbytes pages whereas other 68K based machines used 4Kbytes pages. We must support such environments. Object Oriented subsystems may also have requirements for consistency that are different from page aligned common needs. Thus, each segment managed by our service has an associated "granularity" attribute, that will help to deal with such requests: a granule is the smallest size of the segment that can be granted to a mapper's client.

6.2 Single Writer / Multiple Readers

One of the ways to insure consistency is to block any read operation on a fragment of a segment while a write operation is in progress on the same fragment. When a write operation has completed, one can unblock the pending read operations after having given them back the new image of the fragment as modified by the write. Reciprocally when a write operation starts it is blocked until all the "in progress" read operation have completed. Thus, at any given time, one may have over the network either **several read-only copies** of the fragment or **one, and only one, write copy** of the fragment.

Here, "read-only copy" means that the access right associated to the page is set to read. If a user wants to write into that same page, the CHORUS VM will handle the "artificial" write fault and will ask (using *mpGetAccess*) the mapper for the "write access right" on the page.

In order to achieve this, one needs to know which **fragments** of an object are on which sites and with which access rights *(Read/Write)*. So, with this information we are able to revoke the appropriate rights and/or data. The write access and/or data will be given to a site that wants to write a fragment that was shared readable between several sites. Similarly the read access and data will be given to a site that wants to read a fragment that was previously writable on another site, or readable on several other sites.

6.3 Server Architecture

The decentralized Server is, in fact, composed of multiple servers: one and only one Global Mapper and one Local Mapper per site. We will illustrate the basic architecture of the decentralized mapper, by describing what happens when a regular file is opened and accessed from two sites. The first site opens the file and writes a page. Then the second site, will open the same file and read that same page.

To perform an open, the PM sends a request to the MiX File Manager which loads the corresponding vnode in memory and generates a capability (named *real capability*) to access that vnode. The File Manager then sends an *mpAttach* request to the **"Global Mapper" (GlMp)** with the capability naming the vnode. The Global Mapper records that a new segment is now becoming active and associates a so-called **coherent capability** with the real capability. This capability is sent back to the OM, and then to the PM. The coherent capability is built in such a way that requests applied to this capability will be received by the Local Mapper of the site where it is used (see "Building a Coherent Capability" below).

When the PM has to perform a write operation, it applies the appropriate CHORUS system call to the capability it has received at open time. The CHORUS Virtual Memory generates an *mpPullIn* request that will be received by the Local Mapper. The Local Mapper will then direct this request to the Global Mapper, which in turn will forward it to the MiX File manager. From that point, the Local Mapper of the requesting site is the owner of the page, and known as such by the Global Mapper (see figure 4).

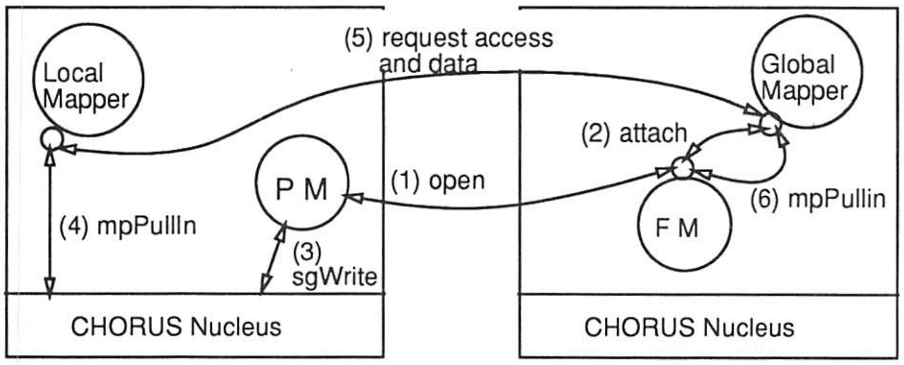


Figure 4. – First access to a file's fragment

Thus, when a second site wants to read the same fragment of the same file, its Local Mapper sends a request to the Global Mapper, which replies that the (probable) owner of the page is the Local Mapper of the first site. Thus, the Local Mapper of the second site is able to request the access and the (up-to-date) data directly from site 1. Before replying, the Local Mapper running on site 1 needs to reduce the access rights of the local CHORUS Nucleus. This is done using the *lcFlush* service, and generates, in turn, a *mpPushOut* of the modified page to the Local Mapper (see figure 5).

A LoMp running on a given site will receive all of the requests sent by the CHORUS virtual memory of that same site. If the local mapper is the owner of the fragment it will be able to

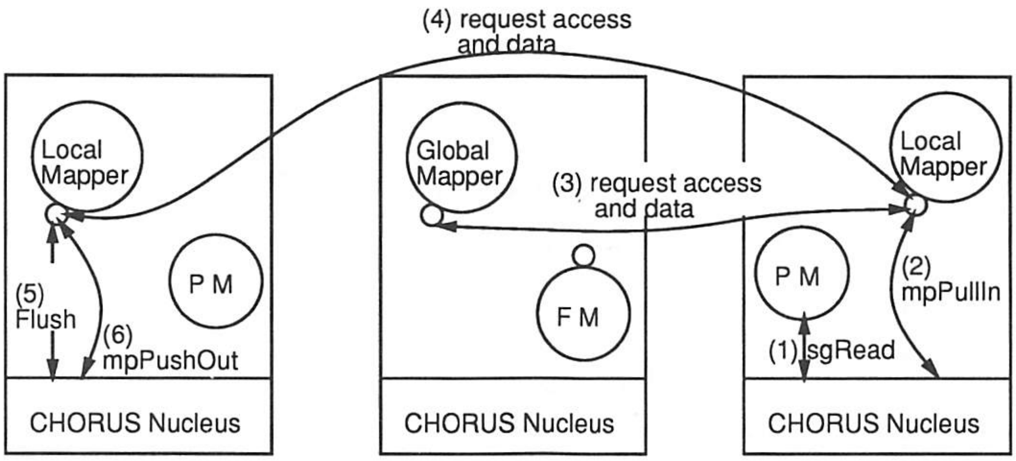


Figure 5. – Concurrent access to a file's fragment

satisfy the request; otherwise it will have to send the request to the last known owner (or probable owner) of the fragment. The GlMp is the default owner of all the fragments of a segment. Thus, if a LoMp doesn't know anything about a fragment, it will ask the GlMp which will either reply with the data or will indicate which LoMp is the probable owner of a fragment.

All fragments have, at any given moment, a single local mapper which "owns" the fragment. A LoMp remains the owner of a fragment until a write access request is received, in which case, ownership is immediately transferred to the new writing site. The local descriptor for the fragment is updated to reflect the new owner. The next access requests received by an old owner will be re-directed to the new owning site. The LoMp has a knowledge of the fragments' probable owner, and the protocol used insures that the request will reach the owner of the fragment after a finite number of tries.

In this way, only the local mappers which share a fragment cooperate in order to maintain its consistency, thus increasing the degree of parallelism. Local mappers use an internal protocol to minimize the number of steps needed to find the owner of a fragment. Our algorithm reduces the number of local mappers to intervene and minimizes the data transfer from one site to another.

6.3.1 Probable Ownership

The performance of the distributed algorithm depends on how efficiently the information on a fragment probable owner is maintained. Fewer steps to reach the fragment owner will be necessary if the probable ownership information is "recent". In order to improve the accuracy of the knowledge of the "probable" owner by the local mappers, the probable owner field is updated with every communication between local mappers.

Figure 6 illustrates a possible access path from a LoMp, which performs a write access request (site 1), to the local mapper which owns the fragment (site 3). In the initial state the fragment information about the probable owner is the following: the Local Mapper of site 1 believes the probable owner is site 2, the Local Mapper of site 2 has its probable owner set to site 3 which is effectively the current owner of the fragment.

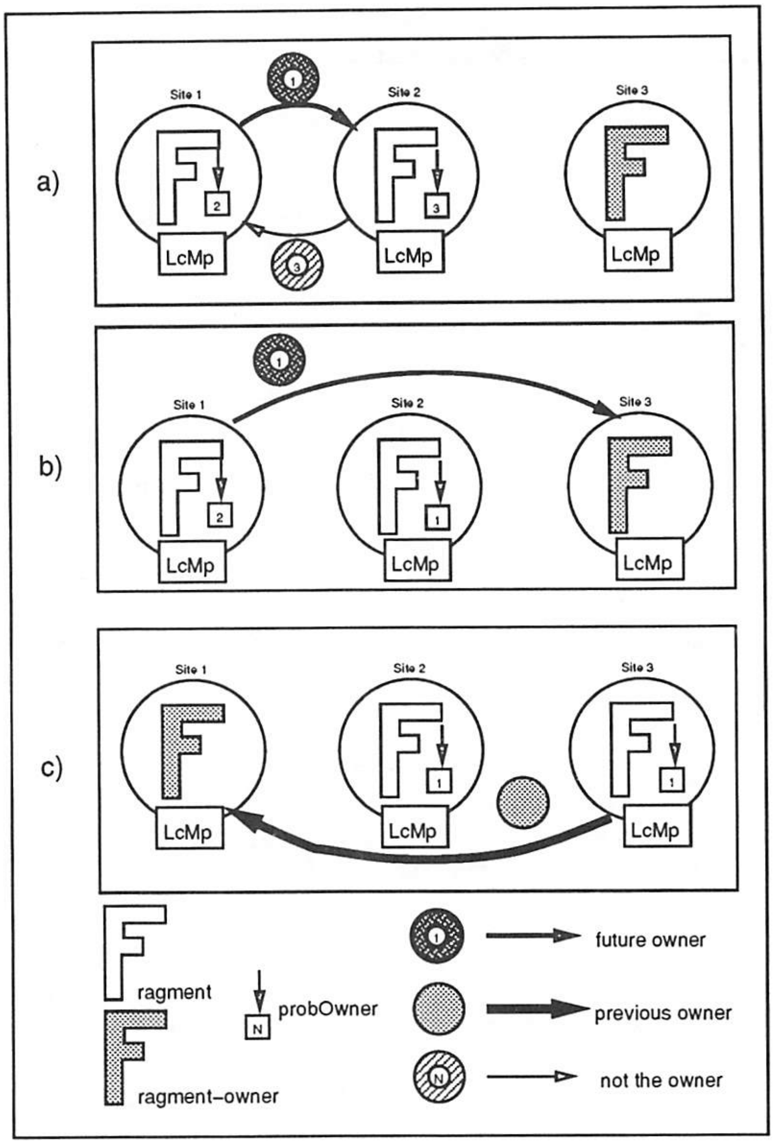


Figure 6. – Searching owner

In figure 6.a LoMp 1 sends a write access request message to LoMp 2 (its probable owner). LoMp 1 will become the owner of the fragment because it asks for write access. Therefore LoMp 2 updates its probable owner field to LoMp1 before replying with LoMp 3 as probable owner. LoMp 1 now knows a new probable owner of the fragment (LoMp 3), and restarts its search until it finds the owner of the fragment (see figure 6.b).

In our example LoMp 3 is the owner of the fragment and the request implies owner change. So, LoMp 3 updates LoMp 1 as probable owner and replies to the the required access. When LoMp 1 receives the reply message from LoMp 3 it, stops the owner searching algorithm and becomes the owner of the fragment (see figure 6.c). The search for the owner is known to be finite (see[Li89a]). For example, after step 8.b, every site requesting a fragment from site 2 will see its request re-directed to site 1 (not site 3) where it will be blocked until site 1 acquires the fragment.

6.4 Some Examples

Rather than describing the entire mechanism we will briefly describe some examples, illustrating how the distributed servers work.

6.4.1 Building a Coherent Capability

The first problem to solve is the naming issue. How to build a capability that will represent the coherent segment? We had one constraint to achieve this goal: the Local Mappers (LoMp) must be able to serve a request for an object they are not yet aware of. This avoids to modify the existing protocols between CHORUS/MiX V.4 servers. Moreover, in a Single System Image environment, processes may freely migrate from one site to another, thus the capability used to access a file in a coherent fashion must be such that there is no need to deal with Local or Global Mappers at migration time, so that migration can be kept as light as possible. This is achieved by constructing capabilities as described below (figure 7):

— The port on which the Global Mapper receives requests, is inserted in a static port group which has a unique 32 bit stamp. The port of the GlMp is the only one to belong to that group. This allows to name uniquely the mapper with a 32 bit long identifier.

— Thus a segment managed by the GlMp will be uniquely identified by its so-called "Global Capability (Gc)" built from the Global Mapper group stamp, and a Unique Identifier of the segment within the Global Mapper (e.g. address of the structure representing the segment).

— Each Local Mapper has a port on which it receives requests either from the local CHORUS Virtual Memory or from other Local Mappers. A group of ports, called the Local Mapper group, contains all these ports. Thus, on any site it is possible to reach the Local Mapper by using the name of that Local Mapper group with a functional addressing mode.

— The capability (Lc) used by a process to access (map, read...) a coherent segment is built by the Global Mapper from: the name of the Local Mapper group, the Unique Identifier of the segment within the Global Mapper and the Global Mapper group stamp.

Any Local Mapper, using this capability is able to rebuild the "Global Capability (Gc)" to send a request to the appropriate Global Mapper for a segment.

Rc	Lc	Gc
OM Porte	Grp. LoMp F	Grp. GlMp
Vnode	UID within the GlMp	UID within the GlMp
Key r	GlMp Stamp.	undef.

Figure 7. – Segment capabilities

6.4.2 mpPullIn

In the *mpPullIn* example (figure 8), the initial state is the following: the fragment data is present only on site 1 and has been modified. In our algorithms the fragment data may be modified only on the site where the local mapper is the fragment owner, the LoMp 1 (i.e. LoMp 3 needs to acquire the write access first).

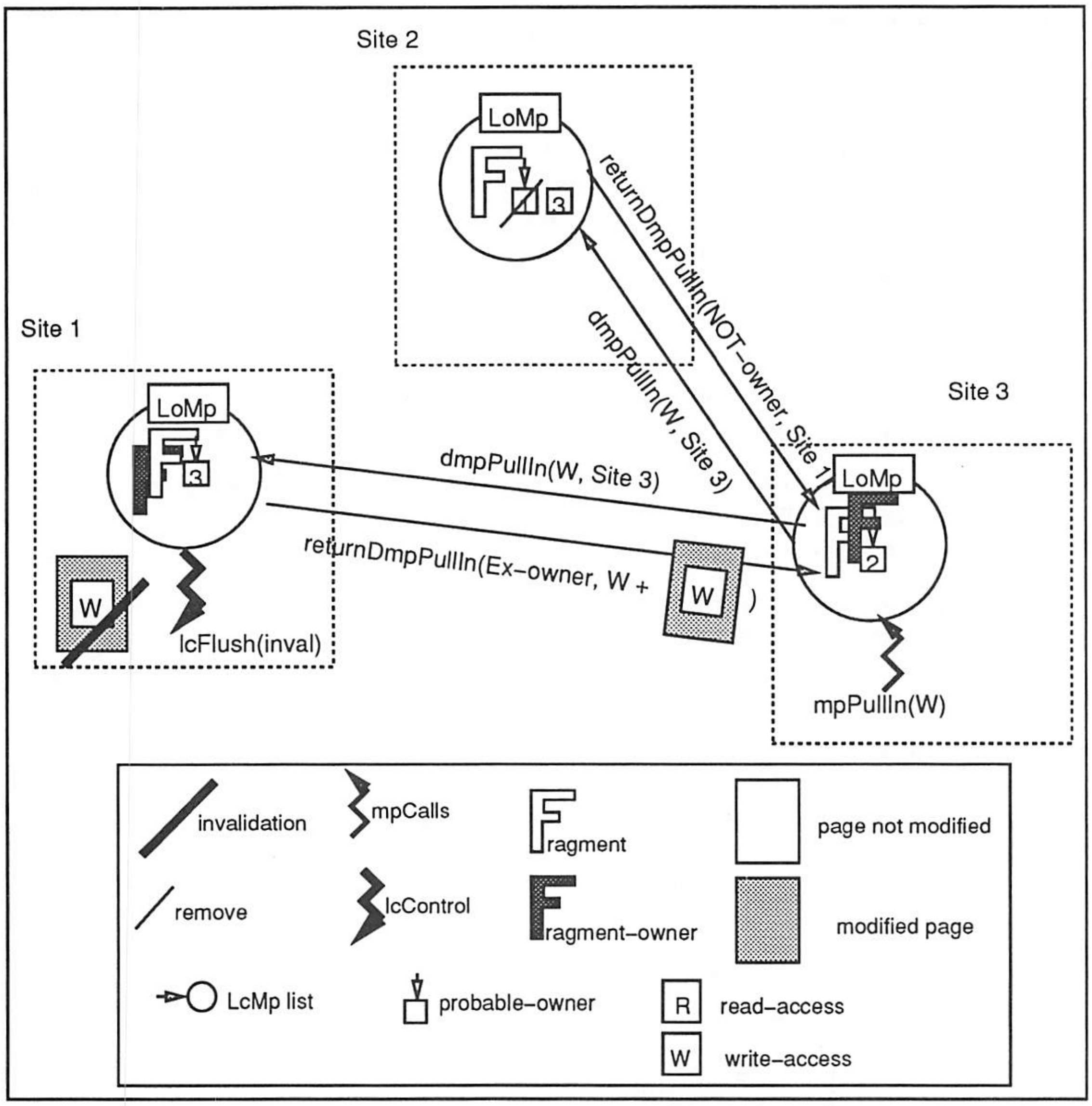


Figure 8. – PullIn example

The virtual memory of site 3 wants to get the fragment data and the write access (*mpPullIn(W)*). The last probable owner known by LoMp 3 is LoMp 2, thus it sends a *dmpPullIn(W, site 3)* request to LoMp 2. *dmpPullIn* is the internal version of the *mpPullIn* request which is exchanged between Local Mappers. This protocol carries extra information allowing LoMp's to update their knowledge of the owner of a fragment.

LoMp 2 is no longer the owner of the fragment so it is not able to give the data and access of the fragment but it replies with its probable owner: LoMp 1. The LoMp of the site 2 updates its fragment owner notion: LoMp 3 because the access required was write. At this moment, the

LoMp 3 must retry the operation with the next probable owner obtained. In this example the LoMp 1 owns the fragment, then it will be able to return the data and the required access.

The LoMp 1 invalidates the fragment in the virtual memory. This operation causes a *mpPushOut* request from the virtual memory to the local LoMp. The LoMp 1 records the new owner of the fragment: LoMp 3 before replying (*returnDmpPullIn*) with the data and the write access.

When the virtual memory needs physical memory space, it performs a *mpPushOut* request in order to write back the fragment modifications before freeing the corresponding physical page. So, it may occur that the data is no longer present (or cached) on the owner's site. In this case the owner LoMp will send a *dmpPullIn* request to the global mapper which will retrieve the data from the real server (e.g. MiX File Manager).

In the *mpPullIn* reply protocol a local mapper can indicate to the virtual memory that the given fragment has been modified but that these modifications haven't been sent to the global mapper. In this case the virtual memory will install the data with the dirty bit set.

6.4.3 Atomicity on large fragment

The local mappers must be able to process requests asking for access rights to a range of several pages. To solve this problem without deadlocks, we insure that pages will be acquired on any site in the same order.

For example, if two sites (I and II) ask for a commo fragment set, say, from page 1 to 3 of a segment, they will try to acquire page 1 first, without holding any other page of the desired set they may already have on their site. Therefore, if site I has get the ownership of page 1, before site II, site II will accept to give up the ownership of any page of the set until it becomes owner of the first page. When site 1 will release ownership on page 1, site II will be able to gain access to page 1, then to page 2 and finally to page 3. It cannot acquire access to page 2 before having gained access to page 1. Similarly, it cannot gain access to page 3 before having gained access to page 2.

6.5 Implementation Details

6.5.1 The GlMp

The GlMp performs segment management: it must maintain only one entry per real capability. The global organization tables are represented in the figure 9.a.

— *Hash segment table*: this is an array of segment list heads accessed via a hash function. Each list is protected by a mutex that is acquired only to walk through the segment list to search, add or delete a given segment. There is only one entry for a given segment.

— *Segment structure*: The segment structure fields are the following:

- *mutex B*: the second level of synchronisation. It is acquired for any operation performed against the segment descriptor.
- *RealCap*: this field contains the real capability of the segment.
- *Ref. counter*: the reference counter maintains the number of "attach" requests performed with the same real capability.
- *Granularity*: the granularity is fixed by the segment (i.e. it is the smallest possible size of a fragment).

  - *head fragment list*: this points to the ordered, disjoint fragment list of the segment that have been requested by LoMp.
  - *head LoMp's list*: it points to a local mapper list. Which contains all local mappers that have requested access to at least one fragment held by this global mapper.

— *fragment structure*: the fragment structure contains the description of a fragment. The components of this structure are the offset, number fragment (offset/granularity) and the access right granted to the probable owner of the fragment. The probable owner is the last LoMp that has requested a write access for that fragment to the GlMp.

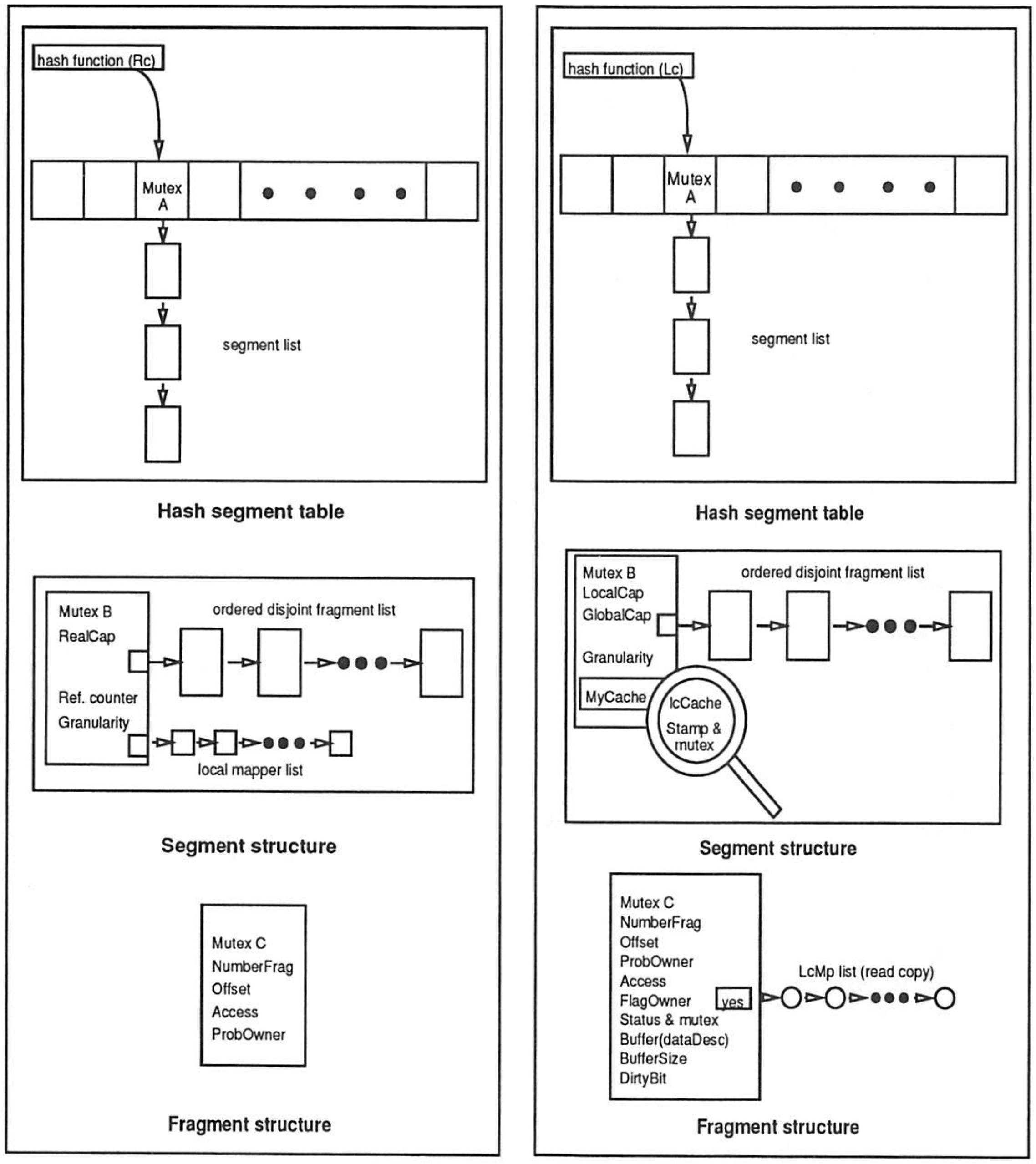
6.5.2 The LoMp

The local mapper structures (see figure 9.b) are similar to the GlMp structures The overall structure is the same but a local mapper must manage other information in order to maintain fragment coherence.

- *hash segment table*: the structure is identical to the GlMp hash segment table.
- *segment structure*: this structure has some different information. Its fields are the following:

  - *GlobalCap*: the Gc (global capability) is rebuilt by the local mapper upon the first request received from the local CHORUS VM, it gives access to the GlMp segment description of the corresponding real segment.
  - *Granularity*: this information is obtained from the Global Mapper during the first *getAccess* or *pullIn*.
  - *MyCache structure*: this structure contains the local "local cache" associated to the coherent segment. There exist only one local cache per segment per site. Due to the preemptive scheduling provided by the CHORUS Nucleus, message passing in CHORUS can not guarantee the soundness of message ordering even though network protocols may insure it (i.e. messages sent from a port **p** to a port **p'** may arrive in a different order than they have been sent). This may cause an inconsistent situation; a *lcFlush* sent after a *mpPullIn* reply may arrive before to the VM. The *stamp* is used to solve a synchronization issue between the mapper and the CHORUS VM, as the VM guarantees to serve messages according to their stamp order.

- *Fragment structure*: this structure contains all information about the fragment.
  - *Mutex C*: this mutex permits requests on different fragment of the same segment to be run in parallel.
  - *NumberFrag*: as we use a fixed granularity per segment we can define a single number per fragment (offset/granularity).
  - *Access*: is the fragment access right granted by the LoMp to the VM for this fragment. It may be WRITE, READ or NULL.
  - *ProbOwner*: this field contains the information of the last owner as known by this LoMp.
  - *FlagOwner*: the FlagOwner's value is "true" when this LoMp is the owner of the fragment else it's "false".
  - *Head LoMp's list*: if the LoMp owns one fragment and there are several "read copies" of it in the network, this list contains the LoMps which have performed a *mpPullIn(R)* or *mpGetAccess(R)* operation on that fragment. This information is used at invalidation time.



a) Global mapper structures b) Local mapper structures

Figure 9. – Distributed mapper structures

- *Status & Mutex*: the Status value describes the pending request. This information is used when a *mpPushOut* request is received to determine whether the data should be written back to its storage (e.g.: *mpPushOut* corresponding to a UNIX sync operation) or sent to another LoMp (e.g. : *mpPushOut* corresponding to a *mpPullIn* operation). This allows to avoid writing back data to its storage when not needed.
- *Buffer(dataDesc)*: Reference to the data being moved that avoids mapping it into the mapper address space, and thus to manage the memory address space at each

mpPushOut/mpPullIn operation.

- *DirtyBit*: this information is recorded at the same time as the body reference (*mpPushOut* time). If the data is "dirty" DirtyBit value is 1 else null. In the *pullIn* reply protocol we can indicate to a requesting LoMp if the data has not been written back after the last modification. The LoMp of the requesting site will install the page as already dirty.

7. Lessons, Status and Future Work

We have adapted the general description of the "dynamic decentralized" algorithms used in IVY to the CHORUS/MiX environment. This basis has been extended to:

— Support for UNIX semantics:
  — It deals with secondary storage issue, allowing data to be written back to the storage server (i.e.: to the disk) or to be moved "dirty" from one site to another.
  — It proposes a scheme to determine the initial location and owner of a fragment, without using a fixed partition mechanism. Such a mechanism would be meaningless in a Single UNIX System Image, where any node can access (or not) any file.
  — Finally, it extends the basic mechanisms to solve the atomicity issue for concurrent read/write operations on overlapping fragments of a file.

— Independent reusable Server:
  — The distributed consistency server is independent from both the CHORUS Virtual Memory and from MiX File Managers. Thus, it could effectively be used in any other environment provided that the protocols are implemented. Due to the notion of granularity associated with a segment, one could potencially use these servers to manage one byte long fragments. In fact, a 2 or 4 byte long fragment could be an integer managed in a distributed consistent way. In other words, our Distributed Consistency server acts mainly as a distributed "token" manager rather than as a Distributed Shared Memory Server.
  — The coherency policy being implemented outside the CHORUS Nucleus itself, this allows some segments to be managed by a "strictly coherent" policy, while other segments may be managed by other mappers according to different schemes.
  — The independence of the server from the nucleus and the subsystem has allowed us to develop it in parallel with the MiX V.4 subsystem, and to test and enhance the CHORUS Virtual Memory interface without being required to implement a full distributed UNIX V.4 system. Once again, micro-kernel and modularity has proven to be an efficient way to develop system software.

The mapper has been implemented and a first version of it is running on COMPAQ 386 machines connected through Ethernet. It is coded in C++. The GlMp is 2500 lines of code and the LoMp is 4500 lines of code. Experiments are also being conducted on a distributed memory multicomputer platform. However, it is still too soon to show any performance figures.

In this first version, we have not paid much attention on the internal mechanisms that manage the ordered disjoint list of fragments. There is still work to be done to manage these lists in an efficient way that does not consume too much memory.
We expect the behavior of the server to fit most of UNIX applications requirements, but in some cases we believe that the scheme we use will not avoid thrashing situations, some futher work will be done to detect and solve these situations.

8. Acknowledgements

We would like to thank Michel Gien and Allan Bricker for their help in reviewing this paper and Bruno Pillard for his editorial assistance. Marc Rozier et Frédéric Herrmann participated in many of the discussions on the design and implementation of distributed shared memory in CHORUS.

9. References

[Abro89a] V. Abrossimov, M. Rozier, and M. Gien, "Virtual Memory Management in Chorus," in *Pro. of the Progress in Distributed Operating Systems Management*, Springer Verlag, Berlin, (18-19 April 1989).

[Acce86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Summer Conference Proceedings 1986*, USENIX Association, (1986).

[Benn90a] Rice University, John Bennett, John Carter, and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," Technical Report, Texas, (Febuary 1990).

[Cher88a] David R. Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333. Vsyst

[Cohn91a] David L. Cohn, Paul M. Greenawalt, Michael R. Casey, and Matthew P. Stevenson, "Using Kernel-Level Support for Distributed Shared Data," in *Proc. of SEMDS II Symp. on Experiences with Distributed and Multiprocessor Systems*, The USENIX Association, Atlanta, GA, (March 21-22, 1991), pp. 247-259. CS/EX-91-216

[Flei89a] Fleisch, Brett and Popek, Gerald , "Mirage: A Coherent Distributed Shared Memory Design," in *Proc. of 12th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, Litchfield Park, AZ, (December 3-6, 1989), pp. 211-223.

[Gray89a] Cary G. Gray and David R. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," in *Proc. of 12th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, Litchfield Park, AZ, (December 3-6, 1989), pp. 202-210.

[Hell90a] SIEMENS, Hermann Hellwagner, "Survey of Virtually Shared Memory Schemes," PUMA Working Paper n° 15, München, Germany, (August 1990), p. 64. CS/EX-90-346

[Li89a] Kai Li and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, (November 1989), pp. 321-359.

[Mull87a] S. J. Mullender, *The Amoeba Distributed Operating System: Selected papers 1984 - 1987*, CWI tract 41, Amsterdam, (1987). AMO87

[Nitz91a] Bill Nitzberg and Virginia Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, vol. 24, no. 8, (August 1991), pp. 52-60. CS/EX-91-287

[Orte91a] Maria Inés Ortega, "Distributed Shared Memory and UNIX: Experimentation in the CHORUS/MiX system," CHORUS Systèmes, (September 1991).

[Pope85a] Gerald J. Popek and Bruce J. Walker, *The LOCUS Distributed System Architecture*, The MIT Press, Cambridge, MA, (1985), p. 148.

[Rozi88a] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, "CHORUS Distributed Operating Systems," *Computing System*, vol. 1, no. 4, (October 1988). CS/TR-90-25

[Stum90a] Michael Stumm and Songnian Zhou, ''Algoritnms Implementing Distributed Shared Memory,'' *Computer*, vol. 23, no. 5, (May 1990), pp. 54-64.

[Tam90a] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber, ''A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems,'' *Operating Systems Review*, vol. 24, no. 3, (July 1990), pp. 40-67. CS/EX-90-398

[Zhou90a] University of Toronto, S. Zhou, M. Stumm, K. Li, and D. Wortman, ''Heterogeneous Distributed Shared Memory,'' Technical Report CSRI-244, Toronto, Canada, (September 1990), p. 31. CS/EX-91-275

Design Considerations for Shared Memory Multiprocessor Message Systems

Nayeem Islam and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue,
Urbana, IL 61801

Abstract

Message passing and shared variables are the two major approaches to interprocess communication. Many distributed parallel programs use algorithms based on message passing because the technique can be implemented efficiently both on distributed and shared memory multiprocessors. The performance of such parallel programs depends on the design of the message passing system.

In this paper, we study experimentally the comparative performance of different message passing system designs on a shared memory Encore Multimax multiprocessor. The systems are measured both by benchmarks and by running example distributed applications. The benchmarks and applications are Intel hypercube programs recompiled for the shared memory machine, but otherwise unchanged. To act as a control, the shared memory machine results are compared with the performance of the benchmarks and applications on the iPSC/2 hypercube. All systems are built using object-oriented techniques in order to minimize the effect of coding different message based systems. The design alternatives considered are buffering, buffer organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space. The results include measurements of the effects of the design alternatives, memory caching, message sizes, copying. Our conclusions are a set of recommendations for improving the performance of message passing systems on a shared memory multiprocessor.

1 Introduction

In this paper, we examine experimentally the design alternatives for message passing systems on shared-memory multiprocessors. We identify a set of orthogonal concerns that parameterize a message passing system and examine system implementation issues for each concern. Benchmarks and applications are used to evaluate the performance of the various alternative implementations. The experiments are performed using message passing systems organized by object-oriented techniques and coded in C++. This approach structures the various message passing system implementations according to specific implementations of each concern, simplifies analysis of the code, and allows the different implementations of each concern to be combined in many different experimental systems with little effort. The measurements from the experiments are analyzed and compared against control experiments. Our results indicate design alternatives which have clear performance benefits and several hardware bottlenecks that impact performance.

The performance of a parallel application depends upon many factors including the algorithm, the parallel programming model, the target parallel computer architecture, the compiler, and the target operating system. Investigation of the impact of operating system design on parallel application performance is difficult because of the work required to make experimental comparisons. Extending an existing operating system to support various parallel programming models using a variety of different implementations without compromising efficiency is a complex and time consuming task. Alternatively, porting different operating systems to the same hardware platform to make comparisons is also expensive. Either approach may produce misleading results. Our research is concerned with building customized operating systems using object-oriented techniques to support high-performance parallel applications. By customizing an operating system and support libraries, we compare the performance of applications for different operating system designs systematically and rapidly.

Message passing and shared variables are the two major ways of interprocess communication on shared-memory or distributed memory multiprocessors. These schemes have much in common. Parallel algorithms based on message passing and on shared variables have implementations on both shared-memory and distributed memory multiprocessors. Some researchers claim improved performance for algorithms if they are written using message based techniques and run on shared-memory machines[23]. There are a variety of ways to implement a message passing scheme on a shared-memory multiprocessor and it is not easy to predict the performance of any one given approach given a description of the machine architecture. In order to understand the issues that effect performance in a message passing system, we compare a spectrum of different message based systems. These systems are compared with benchmarks and applications used to measure and compare distributed memory multicomputers known as hypercubes. In this respect, our work is different from recent schemes that are mostly aimed at improving multiprogrammed thread performance [2]. We are interested in operating system support for shared-memory machines that allows for application performance similar to that on dedicated distributed memory machines such as the Intel iPSC/2 hypercube [12]. Without dedicated resources, applications do not attain the performance of distributed memory machines. Our focus is on effectively running scientific applications in a general purpose operating system.

We simplify the coding of the different message systems by implementing them on *Choices*, an object-oriented operating system. *Choices* is a symmetric multiprocessor operating system written almost entirely in C++. We use object-oriented programming in C++ to track the reuse of code implementing similar design components in different message passing systems. Class inheritance documents the reuse of such code and reduces programmer overhead. Different subclass specializations of particular abstract classes document design differences. The resulting class library captures all of the design differences and similarities between the messaging systems explicitly. Further, the class hierarchies in the library form a framework for assembling new message passing systems built by reusing the components we have tested. This approach allows us to experiment with different styles of message passing quickly by reusing existing code in new designs in a systematic manner.

We present performance measurements for running several applications, including a ring message passing program commonly used for benchmarking message latency, a Fast Fourier Transform (FFT)[25] adapted to the hypercube, and a Simplex program that has been analyzed in detail[33]. Six different message systems are compared using the message latency benchmark. We further investigate the effects of message data transfer on the scalability of the FFT and Simplex applications. The behavior of the application using the various message passing implementations can be understood in terms of copying and the hardware data and instruction cache. The shared-memory machine used in the experiments is an Encore Multimax [13] with NS32332 processors. To act as a control for our experiments, the performance of the applications is compared with their performance on Intel's NX/2 operating system [27] running on the Intel iPSC/2. The results show that the relative performance of the applications on the shared-memory machine is not degraded by either the machine, *Choices*, or the object-oriented techniques used.

Our work differs from existing work in that we use object-oriented design techniques to compare systematically a number of different design alternatives in the implementation of an existing message system protocol. We identify the pitfalls involved in designing a message passing system on a shared-memory multiprocessor. Our message passing protocol is based on the Intel NX/2 hypercube message passing system. The implementation uses recent results from scheduling [6] and spin-locking [1, 17] algorithm studies. We perform a set of control experiments that show that applications running on our message system have similar behavior as on the hypercube. Despite the extensive use of C++ virtual functions in the experiments, the evidence indicates that an object-oriented operating system can be customized to achieve comparable performance to a proprietary operating system supported with purpose-built hardware.

2 Background

In this section, we examine current issues in interprocess communication and object-oriented system design. We review the message passing system protocol, operating systems, and machines that are used in the experiments.

2.1 Parallel Message-Based Applications

Parallel applications running on shared-memory multiprocessors like those built by Alliant, Encore and Sequent and message-based parallel applications running on distributed memory multicomputers like the Intel iPSC2 hypercube[29, 7] have demonstrated notable performance improvements over their sequential counterparts. How well interprocess communication and processor utilization scale as the number of available processors increases is a research issue[36]. The various parallel programming techniques used to program applications for different computer architectures using dissimilar operating systems make comparisons difficult.

There has been little previous research examining the performance of parallel applications running on multiprocessor object-oriented operating systems. To date, few operating systems have been coded extensively in an object-oriented programming language. Operating systems with messaging primitives on distributed memory systems include V System[11], Amoeba[34], and NX/2[27]. Operating systems with shared-memory primitives on shared-memory machines include Umax[13] and Mach[28]. Demos[3] and Mach[28] are examples of systems that implement messaging primitives on top of a shared-memory multiprocessor. Shared distributed virtual memory[22] allows a shared-memory application to run on a distributed memory computer. Various systems like Amoeba[34], Clouds[14], Chorus and SOS[32] are object based. Object-oriented languages have been used with some success to implement parallel processing thread packages such as Amber[10].

Studies of message-based applications by LeBlanc [21] and Lin [23] on shared-memory machines reveal interesting problems and issues. LeBlanc uses different implementations of the same application and concludes that the message-based applications scale better for a small number of processors (less than 64) but for a large number of processors the shared-memory applications scale better. Snyder and Lin also rewrite applications to compare shared-memory and message passing programming models. They report experiments in which contention for memory in a shared-memory multiprocessor appears reduced if the programming model is message-based leading to better performance. They reason that this is caused by locality and large computational granularity of such programs.

The performance of a parallel application is sensitive to the scheduling mechanisms and policies of the operating system [6, 36, 26, 19]. NX/2 allows an application process to be dedicated to a processor for the duration of the application[27]. Unlike NX/2, the Medusa operating system implements coscheduling [26], a technique that *tries* to schedule the processes of an application to run at the same time. The Alliant scheduler[19] supports a fixed number of scheduler classes and uses a scheduling vector for each processor to indicate which classes should be searched for work in what order. Groups of processes are assigned to a scheduler class and each group within a class is run simultaneously. Blocking a particular process blocks the group of processes. Process control and scheduling issues on the Encore Multimax[36] indicate that applications can be written to negotiate with a server for an allocation of a number of processors. The server and the application together determine the level of application concurrency. In a distributed system, techniques have been proposed to locate idle processors efficiently[35, 11].

Bershad [5, 4] has shown that fast interaddress space communication is possible on shared-memory machines. Such schemes can be implemented in the kernel or in shared-memory user space. Our work is similar in that we have implemented the NX/2 messaging primitives both in shared-memory user space and in the kernel. However, we use kernel-based context switching, object-oriented techniques for the design of the facilities and emphasize reuse. Bershad's work concentrates on shared memory machines, multiprogramming, and a simple set of message primitives. The work has not yet been applied to complex message passing systems such as found in the NX/2. It is not clear how a typed, asynchronous message primitive would be encoded efficiently using his techniques. Last, Bershad does not report measurements of any parallel hypercube-like applications using his message system. Our goal was to build a high performance message system that could run existing code. The NX/2 primitives are very general and our experiments involve substantial applications that use most of the primitives.

Our work differs from previous work in that:

1. We identify a larger set of orthogonal parameters that may alter the design of a message passing system.
2. We evaluate our work using primitive benchmarks, as well as real applications.
3. We use object-oriented design techniques.
4. We experiment on an object-oriented operating system.
5. We compare our system with a distributed memory machine that is designed for message passing.

2.2 The NX/2 Message Passing Protocol

The NX/2 operating system includes a set of messaging passing primitives for asynchronous and synchronous communication[27]. In the hypercube, message transfer is reliable but messages may be received out of order.

Two calls implement synchronous message passing:

1. *csend(type,buffer,length,node,pid)* and
2. *crecv(typesel,buffer,length).*

The *csend* call does not return until the message has been put on the network and the buffer specified in the message by the application is free to be used again. *csend* does not necessarily wait for a corresponding *crecv* to occur at the destination process. The *crecv* call selects an incoming message by type and receives it in a buffer. This call is blocking, it does not return until the message is in its buffer.

The corresponding asynchronous calls are:

1. $messageId$ = *isend(type,buffer,length,node,pid)* and
2. $messageId$ = *irecv(typesel,buffer,length).*

The *isend* call starts transmission and then returns but the buffer cannot be reused immediately. A $messageId$ is returned that can be used to query the message passing system periodically to determine if the buffer is free to be reused by the application that invoked the *isend* call. There is no way to query the system to determine whether or not the message was actually received. The *irecv* call registers a request with the message passing system to receive a message into the buffer specified. The call returns immediately with a $messageId$. This $messageId$ can be used to check if a message has been put into the buffer. The *irecv* call can be used to reduce message handling overhead since it informs the message passing system where to put the message when it arrives.

2.3 Choices

Choices is a multiprocessor operating system designed as an object-oriented system using object-oriented coding techniques in the C++ language [9]. The many advantages of an object-oriented approach are reported in [24, 20]. *Choices* has, as its kernel, a dynamic collection of objects. System resources, mechanisms, and policies are represented as instances of classes that belong to a class hierarchy. The system has over 300 classes and 78,000 lines of source code. *Choices* runs on bare hardware: the Encore Multimax, the Apple Macintosh IIx, the SUN Sparcstation 2, the AT&T WGS-386, and the IBM PS/2. It supports both uniprocessor and multiprocessor architectures.

Our goal in the design of *Choices* is to allow the system to be specialized to support parallel processing primitives, different machine architectures, and different application programming models. By using inheritance, encapsulation, delegation, frameworks and other object-oriented techniques, *Choices* is intended as an easily modifiable testbed that can give comparable results for diverse applications running on a variety of computer architectures[30, 9]. All entities in the operating system are modeled as objects and include: classes, system processes, user processes, regions of memory, files, and hardware devices like CPU's and disk controllers.

Fundamental to the design of *Choices* is the notion of frameworks. Each subsystem, for example the virtual memory subsystem, is designed within a framework. A *framework* lies at the heart of any complex object-oriented implementation[38]. It consists of a set of class hierarchies and a design for how the instances of the classes from these hierarchies can be combined to make systems. The key ingredient in a framework is the identification of the abstract concepts of the problem domain rather than an actual implementation[15]. By using the framework with different implementations, it is possible to gain confidence that the abstract concepts model the problem domain closely.

In designing a subsystem such as virtual memory, a framework is proposed, an example implementation is constructed within that framework, and the framework is modified by the practical implementation. New example implementations are created and the framework is updated as necessary. The frameworks evolve by prototyping until they adequately describe the abstract properties of the subsystems that are being built.

In our experiments, we added a message subsystem to *Choices* based on the the NX/2 message passing protocols. The subsystem is implemented within a message system framework. The object-oriented design of this framework is presented in [8]. In this paper, we concentrate on the properties of the message system and how they effect the performance of the message passing system. The framework supports various message system implementations. In addition, we augmented the *Choices* process management, scheduling and application interface frameworks in order to support efficient message based parallel applications.

2.4 Encore Multimax

The Encore Multimax 320 is a shared-memory multiprocessor. NS32081 coprocessors provide floating point processing. Each processor accesses memory through a 64K byte cache and a 100 Mbytes/sec bus. Cache accesses do not invoke processor wait states for main memory access and do not impose any Nanobus (the processor/memory bus) traffic. The cache is thus much faster than the main memory. Maintaining locality of reference to data in the cache is an essential part of achieving high performance[18].

2.5 Summary of Equipment used in Experiments

Our experiments were conducted on a 16 processor Encore Multimax with 32 megabytes of memory and a 16 node Intel iPSC/2 hypercube with 4 megabytes of memory per node. The details of the processors for each system are shown in Figure 1.

Comparison of Processors Based on Manufacturers Specifications					
System	Processor	Clock Rate	MIPS	OS	Messag Passing Primitives
iPSC/2	Intel 80386	16 MHZ	4	NX/2	Written largely in assembler
Multimax	NS32332	15 MHZ	2	*Choices*	C++

Figure 1: Comparison of processors

3 Design Considerations for Message Passing Systems

The hypercube message system primitives were added to *Choices* as a specialization of a framework for message systems. The modifications include message system extensions for the NX/2 message passing primitives, gang scheduling, libraries and several small utility programs. Several changes were required to other subsystems to port hypercube applications to *Choices* without source code modification. The application interface on *Choices* differs from that of the NX/2, requiring the construction of a compatibility library. The NX/2 message passing primitives are defined in *Choices* using a set of abstract classes that provide a "standard" interface. Specific message system implementations are written as concrete subclasses of these abstract classes. This simplified configuring *Choices* for each different message implementation to measure its performance with the benchmarks. This section describes some of the many design alternatives for implementing the NX/2 message primitives on a shared memory machine.

3.1 A Framework for a Message Passing System

The message system framework accommodates different message passing semantics, buffering, and implementation schemes for the NX/2 message primitive interface. Seven factors dictate the construction of a message system and they are represented as separate classes within the framework. These factors are:

- *Location:*
  - *User space*: the message system is implemented in user level. A send or a receive does not involve passing data through the kernel but is implemented using user shared memory. There is minimal checking of parameters and objects passed between processes.
  - *Kernel:* the message system is implemented in the kernel. Message parameters are checked by the kernel.
- *Context Switching: Choices* provides variable weight context switching. Context switching between one lightweight process and another in the same virtual memory space has very little overhead[31]. Context switching between heavyweight processes residing in separate address spaces incurs considerably more overhead. All context switching is implemented in the kernel.
- *Message/Process Interaction:*
  - *Asynchronous*: When a message is sent, the process does not wait for the message to be delivered to the buffer of the receiving process. A copy of the message is made and the kernel returns from the call immediately. When a process attempts to receive a message of a particular type, if the message is not in its port, the receive returns immediately with an *identifier* that the receiver process can later use to get the message. If the message is in the port the receiver receives the message immediately.
  - *Synchronous*: When a message is sent, the process blocks until the system can send the message to the receiving process. When the receiving process receives a copy of the message the sender is unblocked. When a process does a blocking receive, it waits for the sender to send the message.
- *MessagePort:* this is similar to a mailbox. A message may be left in a mailbox. Typically, a mailbox has one receiver at any particular time and multiple senders. A message port has a queue structure associated with it.
- *Transport*:
  - *Process:* an independent process copies the message from source to destination.
  - *Buffered:* the receiver process incurs the overhead of message transfer. When performing a receive the kernel will search the queues for relevant messages.
- *Synchronization*:
  - *Semaphore*: semaphores are implemented in *Choices* by blocking the process and not by busy-waiting.

- *Spinlock*: this is implemented using the following sequence: 1) read the spinlock variable into the cache, 2) while the spinlock is locked, spin read the variable, and 3) when the value changes try to acquire the lock. If the lock is not acquired, the algorithm is retried. Reading the value into the cache reduces bus contention. A disadvantage of the algorithm is that when the lock is freed, many processes may attempt to acquire the lock causing memory contention[1].

- *Transfer:*
  - *DoubleCopy:* the message is copied into a kernel buffer and then into a user buffer.
  - *SingleCopy:* the message is copied once from the sender buffer to a receiver buffer in a shared memory region. This is referred to as restricted message passing[37].
  - *PointerTransfer:* buffer pointers are exchanged but data is not copied. One process passes a pointer to a shared data region to another process. The receiver must dereference the pointer to get access to the message data.

3.2 Implementation Issues

In this section, we focus on the queue management alternatives for ports, the memory management scheme for buffers, the user level message system implementation.

MessagePort Queues In all of the implementations of the message passing system, each port has a message queue. A message is queued because the destination process may not be ready to receive the message. Each message queue element has an associated data segment. Senders and receivers concurrently accessing the queue can cause contention. We describe two implementations that reduce contention:

1. The *send* method adds a message to the head of the message queue. A send acquires the current index of the head of the queue, modulo the length of the queue from an index allocator. The index allocator atomically increments and returns the value of the head of the queue. The send operation acquires a different index every time it is invoked by a sender process and can update the queue in parallel. The critical section for the index allocator is extremely small (a few machine instructions).

   The *receive* method takes messages from the tail. The tail pointer is incremented. If the head and tail of the queue are equal (we contend that this is rare) then a sender is blocked until they are no longer equal. This can be avoided by requiring the queue length and number of cooperating processors to be equal, and by disallowing asynchronous sends.

2. Alternatively, previous message queue entries are reused once the messages in them have been received. This method invalidates the cache less often than the circular queue scheme but results in much more contention for acquiring a reused queue entry. Each entry requires its own lock.

The first alternative may continuously invalidate the cache, particularly for large messages. The second alternative may cause excessive contention for queue locks. We chose the first method because it was similar to that used by [16, 17] on shared memory architectures. However, our implementation does not require a lock per entry since we only allow one receiver and multiple senders per MessagePort. Multiple receivers per queue would necessitate the use of a lock per queue entry.

Message Buffer The kernel keeps preallocated buffers for the double copy version of the message system. This eliminates the overhead of calling the kernel allocator at message delivery time. For the single copy and pointer message systems data buffer areas are dynamically allocated as part of the application. The allocator for shared user memory is synchronized to permit concurrent operation.

User Level Implementation Recent work shows that interaddress space communication primitives can be implemented very efficiently at user level [5]. Such implementations may be particularly appropriate for applications that are tightly coupled, as is the case of the hypercube applications. In *Choices* we designed and implemented a user level message system library.

The user level message system contains shared data (message queues, spinlock variables) and private data (node identifiers). The user level message system has two types of methods: methods that access shared data and methods that access private data. To implement the message passing system, memory allocation primitives are needed to allocate shared memory space and per user private memory space.

4 Benchmarks

In this section, we describe the benchmarks used to evaluate the performance of our message passing system. The first benchmark is a *message latency* benchmark which we refer to as the "ring" benchmark. The message latency of a message passing system is defined as the time for one process to send a message to another process, for that other process to receive that message and send it back, and for the original process to receive it. The ring benchmark is often used to measure message performance on the Intel iPSC/2 hypercube and uses a circular connection between processes. We vary this basic test by accessing the data that is sent between processes in the ring. Whether or not the data is referenced or written can modify message latency.

The second benchmark is a parallel version of the Fast Fourier Transform (FFT) [25]. The Fast Fourier Transform is a computational technique that is used extensively in branches of engineering. In the parallel version of the FFT, the application uses P processors. For a data set of n, the application has $log_2(n)$ butterfly stages, $log_2(P)$ of which are performed over the network. The $log_2(P)$ network stage involve the transfer of $16 \times n \times log_2(P)$ bytes. The FFT experiment is interesting as a scalability study. As the size n of the data increases, the message size increases linearly with n, but the computation increases as $nlog_2(n)$.

The third benchmark is the Simplex linear optimization program. We chose this code because its behavior had been studied extensively on the iPSC/2 hypercube by Stunkel [33] and includes non-trivial, data dependent communication patterns.[1]

In our parallel Simplex algorithm, each processor repeatedly finds a minimum among its local data values and then participates in the identification of a global minimum, the pivot element. The processor that contains the minimum value broadcasts a matrix row or column, depending on the data partition chosen, to all the other processors. Thus, the pattern of interprocessor communication is data dependent.

5 Performance Experiments

The experiments can be divided into two sections: message latency, and application performance. Similar experiments on an Intel hypercube are used as a control. First, we measure message latencies for different shared memory implementations of the hypercube message passing scheme using the ring message benchmark. We also present how these results change if applications read or write the message data. Next, we measure the performance of the FFT and Simplex application benchmarks using the different message passing implementations.

In each of the benchmarking examples, we compare our implementation of the message passing primitives on a shared memory machine with the NX/2 operating system implementation of them running on the Intel iPSC/2 hypercube. Since several benchmarks are run under *Choices* on the Encore Multimax without modification, the corresponding hypercube measurements act as a control for the Encore Multimax implementations. We compare message latency, effect of message data access, application performance and message system overhead. The comparison ensures that we have implemented an efficient message passing system.

[1] Recall that linear optimization algorithms seek to find a solution to an underconstrained linear system that maximizes (or minimizes) some function subject to a linear constraint.

5.1 Message Latency

The ring benchmark measurements allow comparison of the performance benefits of each of the various design alternatives introduced in section 3.1.

Experiments versus Parameters					
Experiment	Location	Transport	Synchronization	Transfer	ContextSwitch
1 ◇	Kernel	Buffered	Spinlock	SingleCopy	Heavy
2 ▽	Kernel	Buffered	Spinlock	Pointer	Heavy
3 ⊗	Kernel	Process	Semaphore	Doublecopy	Heavy
4 ×	Kernel	Buffered	Semaphore	Doublecopy	Heavy
5 ⊕	Userlevel	Buffered	Spinlock	SingleCopy	Light
6 ○	Kernel	Buffered	Spinlock	Doublecopy	Heavy

Figure 2: Parameters measured in ring experiment

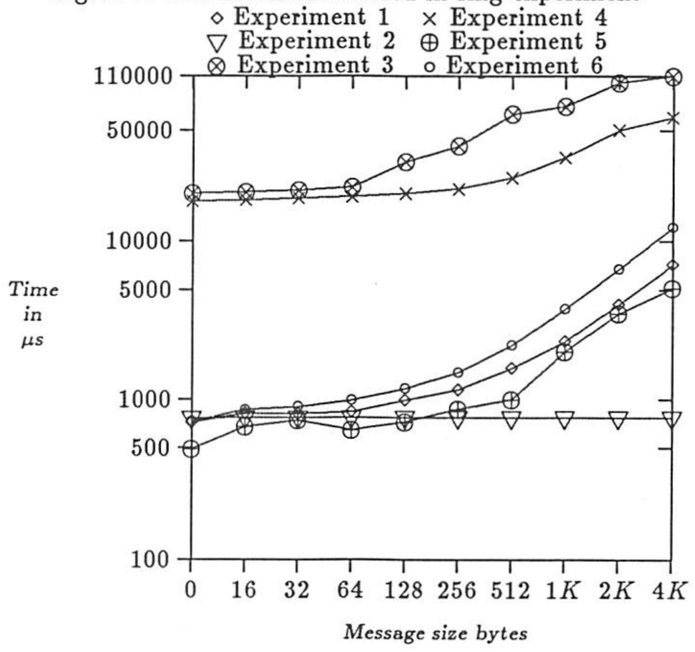


Figure 3: Round trip message times under *Choices* on the Multimax

Figure 2 is a representative list of the experiments evaluated using the ring benchmark. Each experiment measures the performance of the ring benchmark when using a message system constructed with the components specified in the figure[2]. Figure 3 shows the performance of each experiment in μs (microseconds) per round trip for different message sizes. The double copy implementation of Experiment 6 most closely resembles the hypercube message passing semantics.

Using Figure 3 and Figure 6 we can compare the round trip times of Experiment 6 with those of the ring benchmark running under NX/2 on the hypercube. The comparison indicates that the performance of the message system used in Experiment 6 is close to that of the hypercube in terms of absolute performance for small message sizes. However, as the message size increases the performance of the *Choices* double copy becomes worse than the hypercube. At large message sizes the time to execute copy instructions dominates the cost of *Choices* IPC. Since the NS32332 is 4 times as slow as the i386, these results are not surprising.

[2] Each round trip experiment was conducted in a loop of 10000 iterations. In addition, the tests were individually run 20 times. The numbers reported are the averages of the tests runs. We found that the numbers did not vary by more than 4.25 % from run to run.

The crossover point is at 512K message data size. In the next sections, we examine these experiments in more detail.

5.1.1 Transfer Semantics

The overhead for transferring message data from one process to another is an important cost in a message system. Three different mechanisms for transferring message data are examined experimentally, each scheme provides slightly different message transfer semantics. In the following discussions, the double copy, single copy and pointer transfer semantics are represented by Experiments 6, 1 and 2 respectively.

Any transfer of data from one process to another involving a copy depends on the rate that data can be moved from one buffer to another. A minimalist copy program for the Encore Multimax consists of a loop that repeatedly reads and writes 4 bytes of data to memory. The Encore Multimax executes approximately two million instructions a second so that a copy could not transfer more than 4 bytes per μsec. The maximum data rate may be decreased by indexing, by loop instructions, and by memory contention since the Multimax is a multiprocessor. Thus, a processor performing a copy will use only a fraction of the bandwidth of the Encore Multimax backplane (8 bytes per 80 ns.)

Figure 4 shows the measured overhead for copying data between buffers on an Encore Multimax. The achieved data transfer rate is approximately two bytes per μs. Loop instructions are avoided by unrolling the copy loop for powers of two up to 128 bytes. This a good byte copy routine on the Encore Multimax. The copy data rate has a major impact on the performance of the single and double copy message system implementations.

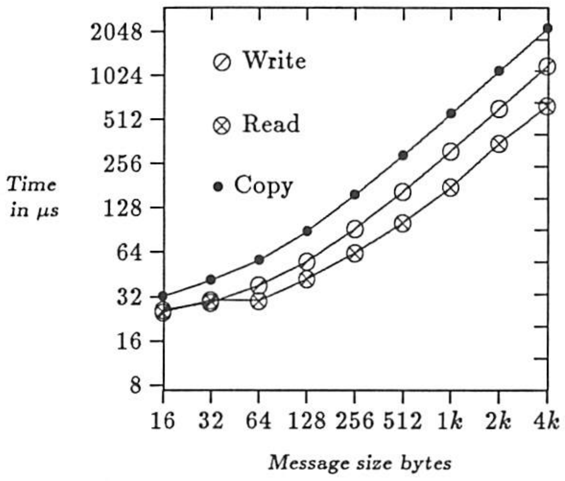


Figure 4: Buffer copy, read and write times on the Encore Multimax

Results Figure 3 shows the message latency between the three message transfer mechanisms. In the pointer implementation (Experiment 2), message latency remains constant as the message size increases. Both the single copy (Experiment 1) and double copy (Experiment 6) implementations suffer copy overhead and have similar performance behavior. As is expected, a single copy implementation is faster than a double copy. The difference between them for a 2048 byte message size is 2690 μs, of which almost 2048 μs can be accounted for by the extra copying of 4096 bytes of data (round trip copying overhead.)

Analysis To explain in further detail the measurements obtained in the roundtrip message benchmark, we examine the operations involved in a message send and receive for each of the transfer mechanisms.

As mentioned earlier, message data copying is a major overhead on the Encore Multimax. On the Encore Multimax, access to memory can be 5 times slower than access to cache.

In the single copy scheme, a process receives a message by copying it from the sender's buffer into its local buffer. When a write memory operation occurs in an Encore Multimax processor, it writes the data to the cache. The Encore Multimax cache is write-through and the message data is written back to the buffer concurrently with further processor operations. The receiving process sends the message back. This requires the new receiving process to read the data (after a write through has occurred) into its buffer. In one complete cycle of the message, the message data is written from cache to memory and read from memory to cache a total of four times. Because the buffers in user memory do not change, the write-through does not have to first read the data in the buffer, see section 2.4. There are $2 \times write + 2 \times read$ operations.

In a doubly copy operation a process sends a message by copying it into a kernel buffer. The process receiving the message reads it from this kernel buffer (after write-through has occurred) and writes it to a local buffer in user memory. This process then reads the local buffer again and writes it to a kernel buffer to send the message back to the original process. The original process then reads the kernel buffer and writes it to its local buffer.

Unless the message data is very large, receiving a message primes the cache with the message data. Thus, copying the message data into the kernel buffer does not require reading the data from memory. The write-through to local memory during receipt of a message can occur concurrently. There are, therefore, $2 \times read_{cache} + 2 \times write + 2 \times read + 2 \times write_{through}$ operations showing that the double copy is slower by 2 cache read and 2 concurrent write-through operations.

Figure 3 shows that the difference between the single copy and the double copy is the time to copy the message data twice. Interference between instruction fetches and data fetches in the cache make more precise calculations very difficult.

5.1.2 Analysis of Hypercube Message Latency in the Control Experiment

Despite the Intel hypercube processor being twice as fast as the Encore Multimax processor, the message latency of Experiment 6 under *Choices* is similar to the hypercube for very small message sizes. As the message size grows to 100 bytes, however, the *Choices* message passing begins to perform better. At message sizes past 512 bytes, the *Choices* message passing system performs worse. For example, a 4096 byte message has a round trip message time of 4360 μs on the hypercube, see Figure 6. A similar message on the Encore Multimax in Experiment 6 has a round trip message time of 12,171 μs, see Figure 5.

The first anomaly occurs because the NX/2 uses a special protocol tailored for short, 100 bytes or less, messages. Above 100 bytes, the protocols change and reserve buffer space at the destination processor by means of additional control messages. As the message size increases above 512 bytes, the hypercube message passing hardware bandwidth compensates for the overhead of the additional messages and the NX/2 message passing system begins to outperform the *Choices* message passing system.

It is interesting to note that the pointer message system of Experiment 1 outperforms the hypercube message passing system for all message sizes. The round trip times for Experiment 1 are a constant 770 μs.

A message is sent between two hypercube nodes by transferring the message data from the memory of one node to the memory of another using the hardware message passing support. There is a fixed communication overhead for re-aligning the message data, generating a header, and processing the special protocol for large messages. The message data is not accessed by a processor and is never copied into a cache. Since the actions of the sender and the receiver are overlapped there are $2 \times read/write_{overlapped}$ operations.

5.1.3 Read and Write Access to Message Data

Actual applications rarely receive a message without accessing its contents. We modified the Experiments 1, 2 and 6 to access the message data. In the read variation of the experiments, the message contents are read before being resent. This forces data into the cache, even for the message system using pointers. In the write variation, the message contents are changed before being resent. Figure 5 shows the round trip times measured by the benchmark for Experiments 1, 2, and 6 with no access, read access, and write access to the message data. For comparison, a similarly modified ring benchmark was run on the hypercube. Figure 6 shows the results measured for the hypercube with no access, read access, and write access. Finally, to account for the behavior of the experiments, we ran a modified copy buffer program on the Encore Multimax to measure the difference between read and write access to a buffer, as shown in Figure 4.

Results Figure 5 shows that accessing the data degrades the performance of all the experimental implementations. In particular, the pointer implementation (Experiment 2) no longer has a constant overhead if the contents of the message are accessed. Figure 6 shows similar results for the hypercube. In general, read access performs better than write access. The reason is partially revealed in Figure 4, which shows that reading data from a buffer is faster than writing data to a buffer.

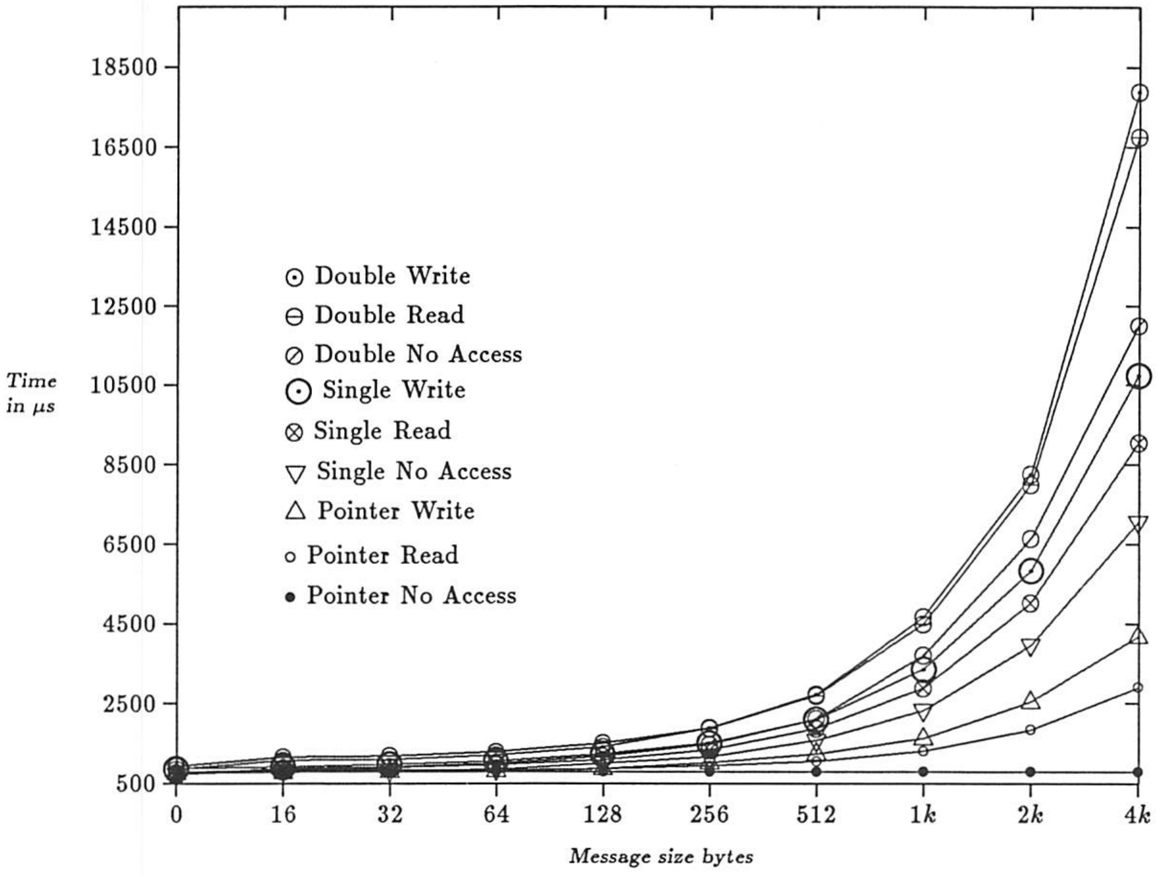


Figure 5: Message passing latency on the *Choices* operating system

Analysis The difference between no access, read access, and write access can partially be attributed to the overhead of reading or writing a buffer. Figure 4 measures the overhead in reading and writing a buffer. For 4096 bytes, a read takes 632 μs, a write takes 1190 μs, and a copy involving both a read and a write takes 1761 μs. These measurements have no loop overhead because the loops are unrolled.

Adding the measured overhead of read or write access to the measured round trip time for a message with no access does not quite account for the round trip times of messages with read or write access. For example, in the pointer experiments the message latency for read access to a message of 4096 bytes is 2884 μs. No access message latency for the same size message is 770 μs. Reading 8192 bytes of data (round trip) should take 1264 μs. Thus, the measurements account for only 2034 μs of the 2883 μs latency. We presume that other factors must be involved including interference between the instruction fetches and data fetches in the cache. The timings for the single copy message system (Experiment 1) and double copy (Experiment

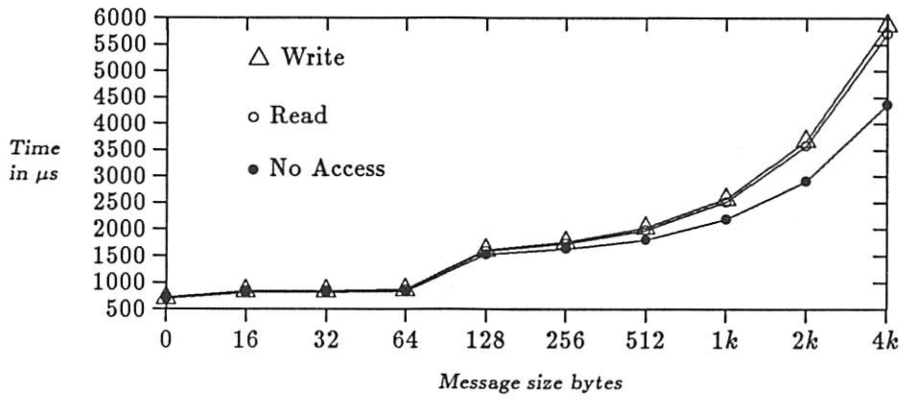


Figure 6: Hypercube message latencies with various types of data accesses

6) are similar to the pointer version. Again, the times cannot be accounted for by a simple computation from our measurements and we presume other factors are slowing down the message system.

5.1.4 Analysis of Hypercube Message Latency with Read and Write Access

Figure 6 shows how the hypercube message latencies change when the data in the message is referenced or written. The behavior is remarkably similar to the *Choices* experiments, particularly Experiment 1 using the pointer message system implementation. However, the pointer message system implementation on the Multimax is still faster than the hypercube message system for the ring benchmark.

The results show that the access time has a significant impact on latency. For example, a 4096 byte message has a round trip message time of 5,700 μs for read access and 5,870 μs for write access on the hypercube. The double copy, write access round trip time is 17,872 μs on the Encore Multimax. The difference in numbers can be attributed to the slower processor on the Encore Multimax. On the hypercube, the roundtrip message latency for a 4K message with no access is 4360 μs as shown in Figure 6. We measured the time to read and write data on the hypercube using a similar program to that used to produce Figure 4 for the Encore Multimax. The time to read 4K data twice is 1300 μs, and the time to write 4K data twice is 1480 μs. Within experimental error, the results show that the additional hypercube message latency is caused by reading or writing the message.

To transfer the data, the system must perform interprocess communication. To access the data, the CPU must read the data into its cache. There are, therefore, $2 \times read/write_{overlapped} + 2 \times read$ operations.

If the message is write accessed, then the message is first read into the cache, changed and then written out with the write-through cache, trickling the effects into main memory. In this case, there are $2 \times read - write_{overlapped} + 2 \times write$ operations.

The worst performance occurs on the hypercube with write access when a message is sent but the receiver is not ready to receive the message. In this case, the message is stored within the kernel of the receiver. This is similar to the double copy message transfer mechanism of Experiment 6. In this case, there are $2 \times read/write_{overlapped} + 2 \times read + 2 \times write_{through}$ operations. The double copy message system on the Multimax has a similar operations except that it has 2 cache reads and 2 writes instead of the overlapped read and write operations.

5.1.5 Synchronization

The synchronization experiment compares spinlocks (Experiment 6) against semaphores (Experiment 4) while keeping other message system parameters the same, as shown in Figure 2.

Results Figure 3, shows that this change increases the message passing latency by a factor of 27 for a null message and a factor of 10 times for a 4K message.

Analysis In *Choices* a process performing a P operation on a semaphore may block. The blocked process is put on a semaphore queue. When a V operation occurs, the first blocked process on the semaphore queue is transferred to the ready queue. In the experiment, *Choices* uses a single ready queue. Processes may run on any processor. A process context switch results in the invalidation of the translation lookaside buffer, page tables and the data cache. A rescheduled process may have to refill the cache with appropriate data. Our experiments show that context switching overhead causes message latency to degrade by a factor of 10 for the ring benchmark.

The *Choices* context switch times [31] are 412 μs on the Encore Multimax for application processes that perform floating point operations. Experiment 6, the spinlock implementation of the message system, indicates that a small message takes about 800 μs for a round trip. Using a semaphore implementation, a round trip causes two context switches. Therefore, we should choose spinning over blocking as is confirmed by the Experiment 3 measurement shown in Figure 3. For large messages, Experiment 6 has a round trip time that is far in excess of 812 μsec. Most of the overhead is in copying the message data. In Experiment 4, a process will block waiting for this copying to complete. As can be seen from the results in Figure 3, for this unusual case the performance of the semaphore implementation does not improve radically as the waiting times increase. These results are different from those obtained by Ousterhout[26] that shows that spinning should be chosen over blocking when the waiting time at the point of synchronization is less than two context switch times[26].

5.1.6 Transport

The process transport experiment compares using an independent process to transport a message from sender process to receiver process (Experiment 3) against having the sender and receiver process perform the task (Experiment 4). The other message system parameters are the same, as shown in Figure 2.

Results Figure 3 shows that the use of a separate process to transport a message degrades the performance of the message system. The effect increases as the size of the message data increases.

Analysis Adding a separate process to deliver messages introduces another process that must be scheduled. This therefore has the potential of increasing the time lost due to cache invalidation and context switching. If one of the application processes is context switched and the message delivery process is allocated to its processor, the cache will be invalidated. However, when the receiver and sender process cooperate to deliver the message as in the double copy system of Experiment 4, they can both exploit the contents of their caches to improve message transmission times.

5.1.7 Kernel Protection

The kernel/user message system placement experiment compares a kernel implementation (Experiment 1) against a user level implementation (Experiment 5) while keeping most other message system parameters the same, as shown in Figure 2. Since context switching does not occur using spinlocks, the "weight" of the context switch is irrelevant.

Results Figure 3 shows that kernel protection for the message system adds a fixed overhead to message latency that does not vary with the message data size.

Analysis Putting the message system in the kernel adds approximately 144 μs to message latency, corresponding to the overhead to make 2 kernel calls. Parameter checking and trap overhead remain a constant as the message size increases.

5.1.8 General Comments

Round trip message times for the Ring Program in μsec			
Number	Activity	Overhead	Total
4	Proxy calls	72	288
4	Process identifications	30	120
12	Virtual functions	6	72
4	Book keeping	40	160
	TOTAL		640

Figure 7: Overhead for null message

In this exposition of different message passing techniques, we have not attempted to produce the optimal message passing system but rather measure the impact of various message passing parameters. The results of Experiment 2 could be improved by combining its pointer transfer mechanism with the placement of the message system in user shared-memory, as in Experiment 5. This would reduce the 770 μs message latency of Experiment 2 by the constant 144 μs. Unfortunately, the advantage of this combination is lost if messages are large and the data in the message are accessed.

The combination of parameters listed as Experiment 6 provide the most accurate reproduction of the NX/2 message passing system. A summary of the breakdown of the round trip time for a null message for Experiment 6 is shown in Figure 7. The component contributing the largest overhead to the round trip time is the kernel call identified in the figure as a "proxy call." This call includes the time to trap into the kernel and copy arguments onto the kernel stack. Moving the message system into user space eliminates this cost. Process identification overhead is extremely high as we do not cache process identifiers in memory. The bookkeeping time refers to the time the message system takes to set flags and any process idle or waiting time for spinlocks. The overhead of programming the message system in C++ corresponds to the virtual function call overhead. Virtual function calls contribute a constant 11.25% to the message latency overhead that is independent of the message data size. Replacing the virtual function calls by C function calls halves the overhead from 72 μs to 36 μs. Thus, the additional cost for programming the message system in C++ is less than 6%. As the message size increases, the use of C++ becomes an increasingly smaller concern.

In the next sections, we will be discussing message systems 1, 2, and 6. We do not consider message passing systems 3 and 4 because they are too slow and we do not consider 5 because we are interested in retaining some degree of protection between applications and we would like to run distributed memory applications with as few changes as possible.

5.2 Application Performance

Simple benchmarks often only reveal some of the bottlenecks that might effect the performance of a system. They must be complemented with measurements of actual applications in order to obtain a good understanding of the performance of a system. In this section we present the results of running applications on three implementations of the message system: single copy, pointer, and double copy implementations. The three implementations were benchmarked in the previous section on message latency as Experiments 1, 2, and 6.

The three implementations differ only in the transfer semantics used for message data. However, as we observed in the last section, this parameter has a significant effect on message latency and should impact application performance. In particular, Figure 3 indicates the pointer version should improve the performance of applications more than either the single or double copy implementations. In the next two sections, we present the results of running two applications, the Fast Fourier Transform (FFT) and Simplex algorithms, on the experimental message passing systems.

5.2.1 FFT Application

The FFT application illustrates how message systems behave as message sizes and numbers increase. The Fast Fourier Transform[25] application was compiled without change and run under *Choices* using the double copy message system that most closely resembles the hypercube message passing semantics. The FFT application had to be rewritten to use the pointer and single copy message passing systems. An effort was made to keep the changes to a minimum and very little code, apart from memory allocation, was changed. In the reimplementations of the application, the data structures that are passed as data in messages are allocated out of shared memory rather than private memory. Figure 8 shows the speedup versus the number of processors for each implementation of the FFT application for a fixed data size. For comparison purposes, the performance of the FFT on the Intel hypercube under NX/2 is also displayed.

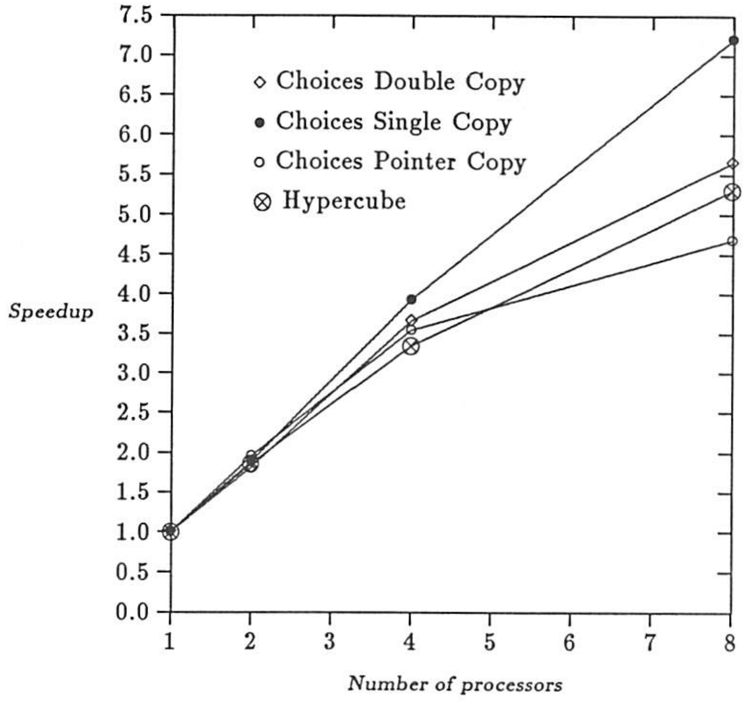


Figure 8: Speedup of 1024 point FFT on *Choices*

Cross comparison of FFT on various implementations			
Number of Nodes	Pointer	SingleCopy	DoubleCopy
1	1	1	1
2	1	1.08	1.04
4	1	0.90	0.96
8	1	0.65	0.83

Figure 9: Cross comparison of 1024 point FFT on *Choices* with increasing processors

Results Figure 9 is a table comparing the execution times of the three implementations for a varying number of processors. The results are scaled relative to the performance of the pointer implementation. The FFT speedup curves for the three variations in message system transfer semantics are identical for 1 processor, indicating that there is no overhead incurred as a consequence of modifying the FFT algorithm to accommodate the changes in transfer semantics. The pointer implementation performs worse than either of the other implementations when the number of processors is greater than 4. Figure 8 shows that the single copy implementation speeds up better than the other implementations, including the hypercube implementation. The 8 processor case seems to indicate the single copy implementation continues to be a better implementation as the number of processors increases. The speedup curves of the application on the Intel hypercube and the double copy shared memory machine implementation are almost identical.

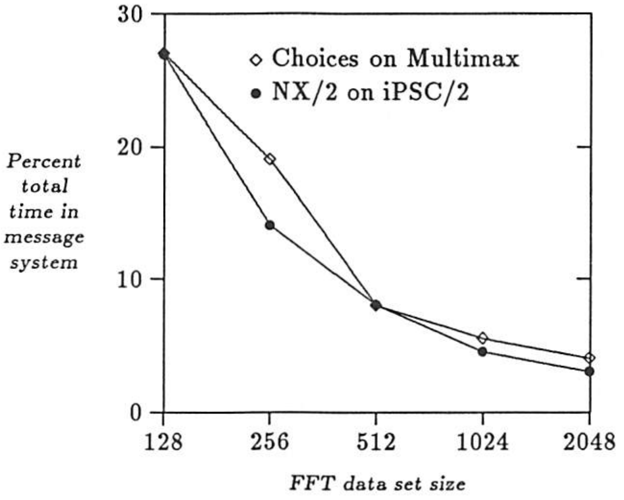


Figure 10: Variation of percentage of time in message system with increasing data size for an FFT running on 4 processors

Analysis The pointer implementation's poor FFT behavior is not predicted by the message latency benchmark results and requires explanation. It is 35% slower than the single copy implementation. Examining the FFT program, we observe that the data space for each processor is allocated as an integral multiple of 64K bytes. The FFT algorithm exchanges messages in a butterfly configuration. Because of this symmetry, the processors in the algorithm access data that are located at addresses that are modulo 64K apart. Each processor is initialized with a copy of the data set which is 16k bytes in size and is aligned on a 64K boundary. The algorithm on each processor receives messages containing pointers to other processor's data regions. The processor uses the same offsets into these data regions in its computations causing comprehensive cache line invalidation. The measured difference in performance between the pointer and single copy implementations is 335,432 μs. If a cache entry is invalidated, it will cause a read from memory when it is accessed. The cache access time is 25 ns but memory access time is 320 ns. Much of the performance degradation of the pointer implementation can be accounted for by cache line collisions. As the number of processors increases, the number of collisions will increase.

Because the processors are executing the same instructions concurrently, there is also a possibility that when the cache entry is reread, there will be memory module contention. Processors in an Encore Multimax can access a single memory module every 320 ns, see section 2.4. In the pointer implementation, 8 processors can access the same module in 2.56 μsecs. Using the data from Figure 4 measurements, a single processor can read 4 bytes of data in a 4K message in 617 ns. If 8 processors read all of the data in a message in a synchronized manner, contention will effectively quarter the performance of reading the message data.

To validate this hypothesis, we changed the allocator to return addresses that were not 64K aligned. The speedup of the pointer implementation increased from 4.68 to 6.26 for the eight processor case. With this modification, the pointer version was 15% (instead of 35%) slower than the single copy implementation.

One difference between the pointer implementation and the other implementations is that the index generation code has to differentiate between high and low indices. We measured this overhead as 20,000 μs.

Factoring out the index generation code additional overhead would increase the speed up from 6.26 to 6.45. Factoring out this overhead and avoiding the cache line invalidation problem, the pointer implementation is still 72,000 μ or 11% slower than the single copy for 8 processors. This remaining difference is most likely due to memory contention and interference in the cache between instruction and data access.

Both the single and double copy message systems rearrange the data by copying it to new locations, avoiding much of the cache line invalidation problems. The extra copying overhead in the double copy implementation makes this implementation perform worse than the single copy.

The FFT Hypercube Control Experiments Results Figure 8 shows that the speedup curve of the hypercube implementation of the FFT is similar to the speedup of the double copy Encore Multimax implementation. Instrumenting the hypercube software and Encore Multimax software allows a comparison to be made of application and system overhead. Figure 10 shows that the proportion of time spent in the NX/2 message passing system for the FFT application is almost the same as that spent in the double copy message passing system on the Encore for varying FFT data set sizes. This confirms earlier results that the overheads for processing a message of a given size using NX/2 and the double copy message systems are similar. In terms of absolute timings, our measurements showed that the hypercube implementation of the FFT has better performance than the Encore Multimax implementation in proportion to the speed of the floating point unit and the CPU.

FFT Hypercube Analysis A detailed analysis of the FFT execution traces on the hypercube reveal that there is a variation between the times of successive interprocessor communications. Although, each processor executes identical application code, the data values differ, and the differing times to evaluate iteratively the trigonometric functions create a communications asynchrony. This application asynchrony results in some unavoidable message passing overhead on both systems.

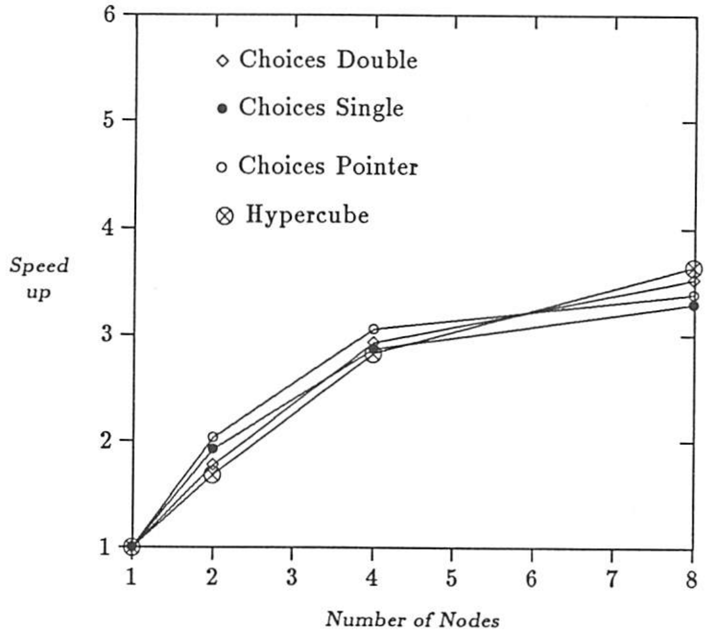


Figure 11: Speedup of the Simplex algorithm on *Choices* and on NX/2

Simplex Application The Simplex application is a non-trivial application and permits a study of how message systems behave for bimodal message size distributions and data dependent communication patterns.

The behavior of the Simplex code has been studied extensively on the iPSC/2 by Stunkel [33]. Again, the Simplex application did not have to be changed for the double copy message system, but did require minor changes for the pointer and single copy message systems.

Cross Comparison of Simplex on Choices			
Number of Nodes	Pointer	SingleCopy	DoubleCopy
1	1	1	1
2	1	1.06	1.20
4	1	1.03	1.05
8	1	0.98	0.95

Figure 12: Cross comparison of Simplex on *Choices* with number of processors

Results Figure 11 shows the speedup of the Simplex application on the various message passing systems including the hypercube. The speedup curves of all the message system implementations are very similar but the ordering of the performance of the implementations changes from that for the FFT. The single copy implementation performs worse than either the pointer or double copy implementations for the numbers of processors shown. The hypercube implementation performs better than any of the shared memory implementations as the number of processors increase. Figure 12 compares the execution times of the three implementations for a varying number of processors. The results are scaled relative to the performance of the pointer implementation.

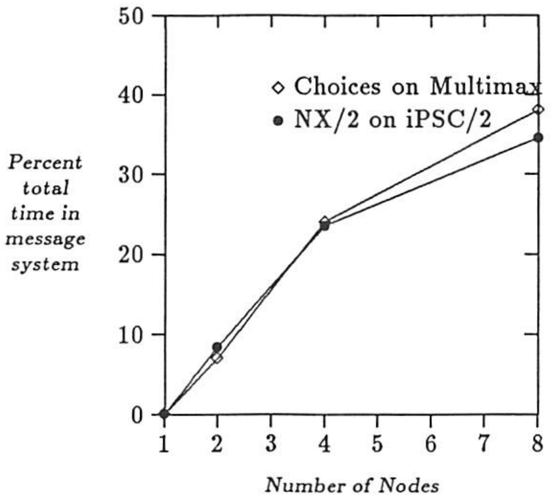


Figure 13: Percentage of time spent in the message system for the Simplex algorithm on NX/2 and *Choices*

Analysis In the Simplex algorithm small messages predominate. The weighted average message size for Simplex is 171 bytes. The fixed overhead for transmitting a message in Simplex is large compared with the variable overhead for copying message data. This has the effect of reducing the differences between the various shared memory message passing systems.

In the Simplex application, pivot distribution is broadcast from one processor to the others. The single copy message system is inefficient at broadcasts because each process receiving a message copies the message data. This causes read access memory contention. The double copy message system is more efficient than the single copy one because the sender process copies the message data to be broadcast from its cache into kernel buffers destined for other processes. These other processes copy the data from their individual kernel buffers and avoid memory contention.

The Simplex Hypercube Control Experiments Results The behavior of the shared memory implementations is similar to the NX/2 hypercube implementation. In both cases, the cause of sub-linear speedup is communication overhead. Figure 13 shows the fraction of time spent in the message passing system when the Simplex code executes under NX/2 on the Intel iPSC/2 hypercube and under *Choices* using the double copy message system on the Encore Multimax.

Analysis As the number of processors increases, the fraction of the time spent in the message passing system increases. There are two reasons for this. First, the total computation is fixed, and as the number of processors increases, the fraction of the work assigned to each processor decreases. Second, the total number of messages sent by all processors increases; most of these messages are small and represent the overhead to find a global minimum among a larger group of processors. Thus, in the case of the Simplex application, because of its large number of small messages, it may be desirable to use a message system in user space.

6 Conclusion

In this paper, we study experimentally the comparative performance of different message passing system implementations on a shared memory multiprocessor using benchmarks and applications. The message passing system is based on the NX/2 hypercube message passing primitives and this allows us to compare the benchmarks and applications on both machines. The comparison acts as a control to ensure that we have implemented the message system efficiently.

All of the software written for the experiment is object-oriented and written in C++ including the message system implementations and the operating system. Object-oriented techniques were used to structure the message system to minimize the coding differences between different message system implementations. The message system design alternatives considered were buffering, buffer and queue organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space.

The results from our study show that different message system implementations can impact the performance of benchmarks and applications considerably. In our experiments, a message system that uses a blocking synchronization primitive ran applications an order of magnitude slower than one that used spin-locks and gang scheduling. Kernel implementations of a message system imposed a constant additional overhead on application performance caused by kernel entry and exit costs. In one example studied, an FFT application sent large messages and a kernel implementation would not impose a significant overhead on its performance. However, a Simplex application sent many small messages and a user-level implementation could improve its performance.

In some applications, a message system based on passing pointers to shared memory locations can be very efficient. In other applications, like the FFT application we studied, pointer passing schemes may result in cache line invalidation and memory contention as the application performs its parallel computation. Message systems based on a single copy or double copy of the message data can perform well for such applications by moving the data to locations where cache line invalidation is reduced and memory contention is eliminated. Message systems based on a double copy scheme impose much overhead on simple applications. In the FFT application, the overhead makes the double copy scheme significantly slower than the single copy scheme.

When there are a large number of small messages in an application, the mechanism used to transfer the data is not as relevant as the fixed cost of sending and receiving messages. The Simplex application sends a large number of small messages and the behavior of the different kernel message passing systems was similar. Memory contention appears to be reduced for broadcasts of a message from one process to others using a double copy message system instead of a single copy or pointer implementation. This effect was noticed in the Simplex algorithm in which one processor broadcasts a pivot point message to all the other processors.

Our recommendations for writing a message system for a shared memory multiprocessor are:

1. It is very unlikely that a particular message system will work efficiently for a large number of different applications. Design the message passing system so that it can be customized for an application.

2. Eliminate context switching by using gang scheduling and non preemptable processes. Context switching is a major overhead on the machines on which we ran our tests. RISC based architectures are likely to make context switching worse.

3. Replace blocking synchronization primitives. For the same reason, synchronization operations which cause a context switch are a major overhead.

4. Organize message queues to eliminate lock contention. Memory contention in the Encore Multimax is a real issue, even for quite small numbers of processors if those processors are trying to access the same lock.

5. Choose an appropriate message transfer semantics for the problem. Some message transfer mechanisms are too heavyweight for some applications that do not require a particular constrained message semantics. Other message semantics are too lightweight for applications and create memory contention.

   (a) Eliminate copying if the message data is not accessed. A pointer transfer message system is quite adequate if the data in a message is not to be accessed directly by many nodes that the message passes through.

   (b) Choose a copy transfer semantics to eliminate memory contention, where necessary. A pointer transfer message system suffers severely from memory contention if the applications are symmetric and use broadcasts. Copy transfer semantics distributes memory accesses, reducing cache line invalidation and memory contention for the same memory location and memory module.

   (c) Optimize the way the cache is used in the transfer mechanism for particular application message passing patterns. Implement broadcasts by copying from a cached copy of the message if possible.

6. Use a kernel-based message system to simplify the porting or development of applications to a multiprocessor. A kernel-based message system provides a better debugging environment.

7. Use a user-level message system for applications that a large percentage of small messages.

8. The percentage of time spent in the message system is similar for both FFT and Simplex on the hypercube and on *Choices* on the Encore Multimax. Memory access appears to dominate the performance of both message passing systems.

9. Use object-oriented programming techniques to simplify customizing a message system for a particular application. The techniques introduce virtual function method lookup overhead, but the overhead is much smaller than the benefits obtained by customization.

References

[1] T.E. Anderson. The Performance of Spinlocking Alternatives for Shared Memory Multiprocessors. In *IEEE Transactions on Parallel and Distributed Processing*, pages 6–16, January 1990.

[2] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory multiprocessors. In *IEEE Transactions on Computers*, pages 1631–1644. IEEE, 1989.

[3] Forest Baskett, J. H. Howard, and John T. Montague. Task Communication in DEMOS. *ACM Operating Systems Review*, pages 23–31, November 1977.

[4] Brain Bershad, T.E. Anderson, Ed Lazowska, and Henry Levy. Light Weight Remote Procedure Call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 102–112, December 1989.

[5] Brian Bershad, T. Anderson, Edward Lazowska, and Henry Levy. User Level Interprocess Communication for Shared Memory Multiprocessors. Technical Report 90-05-07, University of Washington, July 1990.

[6] David Black. Scheduling support for concurrency and parallelism in the Mach Operating system. *IEEE COMPUTER*, pages 35–43, May 1990.

[7] David Bradley. First and second generation hypercube performance. Technical Report UIUCDCS-R-88-1455, University of Illinois Urbana-Champaign, September 1988.

[8] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In Luis-Felipe Cabrera and Vincent Russo, and Marc Shapiro, editor, *Object-Orientation in Operating Systems*, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[9] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[10] F.G. Chase J.S., Amador, Levy HM Lazowska E.D., and Littlefield RJ. The Amber Sytem: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 28–50, December 1989.

[11] David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, March 1988.

[12] Paul Close. The iPSC/2 Node Architecture. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, January 1986.

[13] Encore Computer Corp. *Multimax Technical Summary*. Encore, Marlborough, Massachusetts, 1986.

[14] P. Dasgupta, R. J. LeBlanc, and W. F. Appelbe. The Clouds Distributed Operating System: functional description, implementation details and related work. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, June 1988.

[15] Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 Programming system*. ACM Press, Cambridge,Mass, 1989.

[16] Jeffrey Goodman, Mary Vernon, and Philip Voest. Effficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–73. ACM, April 1989.

[17] Gottlieb, Lubachevsky, and Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. In *ACM Transactions on Programming Languages and Systems*, pages 164–189. ACM, April 1983.

[18] Mark Hill and James Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the ACM*, pages 97–102, August 1990.

[19] Herb Jacobs. A User-tunable Multiple Processor Scheduler. In *1986 Winter USENIX Conference Proceedings*, pages 183–191, January 1986.

[20] Gary M. Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale, Florida, October 1989.

[21] T.J. LeBlanc. Shared Memory versus Message passing in a tightly coupled multiprocessor:A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466. ACM and IEEE, August 1986.

[22] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[23] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International conference on parallel processing*, pages 163–170, August 1990.

[24] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, pages 65–79, Denver, Colorado, October 1988.

[25] H. Miyata, T. Isonishi, and A. Iwase. Fast Fourier Transformation using Cellular Array Processor. In *Parallel Processing Symposium JSPP 1989*, pages 297–304, February 1989.

[26] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International conference on Distributed Computing Systems*, pages 22–30, July 1982.

[27] Paul Pierce. The NX/2 Operating System. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, January 1986.

[28] Richard Rashid. Threads of a new system. *UNIX Review*, August 1986.

[29] Daniel Reed and Richard Fujimoto. *Multicomputer Networks - Message based Parallel Processing*. The MIT Press, Cambridge, Massachusetts, 1986.

[30] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[31] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[32] Marc Shapiro. The Design of a Distributed Object-Oriented Operating System for Office Applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[33] Craig B. Stunkel. Linear optimization via message-based parallel processing. In *Journal of Parallel and Distributed Computing*, pages 264–271, August 1988.

[34] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[35] Marvin Theimer. *Preemptable Remote Execution Facilities for Loosely Cooupled Distributed Systems* . PhD thesis, Stanford, June 1986.

[36] A. Tucker and A. Gupta. Process Control and Scheduling issues for Multiprogrammed Shared Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 159–166. ACM, December 1989.

[37] S-Y Tzou and David Anderson. A Performance Evaluation of the Dash Message Passing System. In *Software Practice and Experience*, pages 1631–1644. John Wiley, 1991.

[38] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Programming. *Communications of the ACM*, 33(9):104–124, September 1990.

EXPERIENCES OF HANDLING MULTIMEDIA IN DISTRIBUTED OPEN SYSTEMS

Nigel Davies, Geoff Coulson, Neil Williams and Gordon S. Blair

Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg,
Lancaster,
LA1 4YR,
U.K.

telephone: +44 (0)524 65201
e-mail: nigel,geoff,nw,gordon@comp.lancs.ac.uk

ABSTRACT

The implementation and use of a platform designed to support distributed multimedia applications is described. The platform provides a programming interface compatible with emerging ODP standards, and uses a transputer based workstation enhancement unit to provide the necessary performance. Implementing such a platform, where the dual concerns of open systems compatibility and performance needed to be addressed, poses a unique set of challenges. The design decisions taken in our implementation are evaluated, and our experiences of both implementing and using the platform are presented.

1. Introduction

This paper describes the results of on-going research at Lancaster aimed at producing a platform to support distributed multimedia applications. The problems of providing such a platform are compounded by the need to provide high-performance (in order to support many of the more demanding media types) whilst working within the proposed ISO ODP (Open Distributed Processing) architectural framework (to support applications in a heterogeneous environment). These problems are described in detail in [Blair,91a].

In order to provide the necessary performance, and to allow existing workstations to manipulate multimedia information we have designed and built a multimedia enhancement unit called the multimedia network interface (MNI) [Ball,90]. This connects directly to the workstation, its associated multimedia devices (e.g. display, camera, speaker and microphone), and a high-speed network as shown in figure 1.

The programming interface for this multimedia environment is based on work from the ODP community and in particular from the ANSA/ISA project [APM,89]. All services, including those on the multimedia enhancement unit, are treated as objects, i.e. encapsulations of state and operations defined on that state. This allows programmers to interact with services in a uniform manner, irrespective of their implementation (hardware or software) and location (on a workstation or on the multimedia network interface). Services may be distributed, in which case communications between services is via remote procedure call.

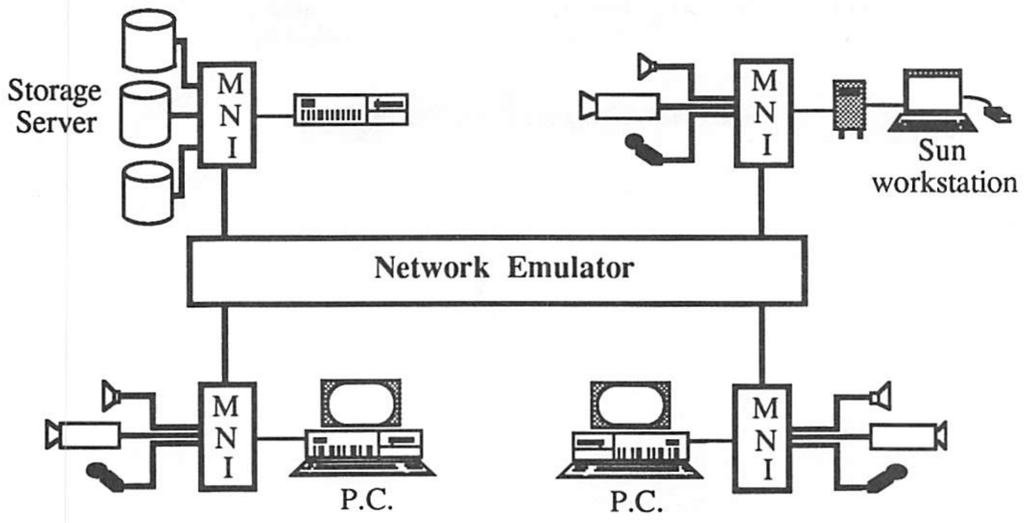


Figure 1: The Experimental Configuration

Section 2 expands on aspects of the computational model described above, and focuses on a partial implementation of this model called *ANSAware* which we have used as a basis for our work. Extensions to the model, its ANSAware implementation and supporting hardware are discussed in detail in section 3. Section 4 presents our experiences of this implementation, and of its subsequent use to develop a number of applications (including a video conferencing facility, a video juke box, and a multimedia document system). Section 5 places our work in context by presenting a comparison with a number of other distributed multimedia systems. Section 6 contains concluding remarks, and section 7 our references.

2. ANSAware

The ANSA/ISA project aims to define a complete framework for the design and construction of distributed systems. The ISA (Integrated Systems Architecture) project is funded within the E.C.'s Esprit program and is derived from the U.K.'s Alvey funded ANSA (Advanced Network Systems Architecture) project. The scope of the architecture being developed is wide and takes on board the full range of issues from overall business objectives to detailed implementation choices. The complexity inherent in this broad view is managed by partitioning the architecture into five viewpoints: enterprise, information, computational, engineering and technology. The focus of our work at Lancaster is on the computational and engineering viewpoints, and these are described in some detail below.

2.1. ANSA Computational Model

The computational viewpoint provides a programming language model of potentially distributed objects and their modes of interaction. In the ANSA computational model, all interacting entities are treated uniformly as *objects*. Objects are accessed through *interfaces* which define named *operations* together with constraints on their *invocation*. Interfaces are first class entities in their own right, and references to them may be freely passed around the system. Interface references are also the sole entities which may be passed to and from operations as arguments and results. Operations may return different combinations of types of interface reference in different circumstances: these are known as alternative *terminations*.

Services are made available for access by *exporting* an interface to a *trader*. The trader therefore acts as a database of services available in the system. Each entry in this 'database' describes an interface in terms of an abstract data type signature for the object and a set of attributes associated with the object. A client wishing to interact with a service interface must *import* the interface by specifying a set of requirements in terms of operations and attribute values. This will be matched against the available services in the trader and a suitable candidate selected. Note that an exact match is not required: ANSA supports a subtyping policy whereby an interface providing at least the required behaviour can be substituted. Finally, once an interface has been selected, the system can arrange a *binding* to the appropriate implementation of that object and thus allow operations to be invoked.

2.2. ANSA Engineering Model

The engineering model sets out specifications, guide-lines and concepts by which an abstract computational model may be realised at a systems level. APM Ltd. have released a software system, known as ANSAware, which is a partial implementation of the computational model. In particular it does not enforce the computational model requirement that all operation arguments and results are interface references: most arguments and results are passed by value. In engineering terms, the ANSAware package is a fairly complete implementation of the ANSA engineering model as described in the ANSA Reference Manual [APM,89].

To provide a platform conformant with the computational model, the ANSAware suite augments a general purpose programming language (usually C) with two additional languages. The first of these is IDL (Interface Definition Language), which allows interfaces to be precisely defined in terms of operations as required by the computational model. The second language, dpl (Distributed Processing Language) is embedded in a host language, such as C, and allows interactions to be specified between programs which implement the behaviour defined by these interfaces. Specifically, dpl statements allow the programmer to *import* and *export* interfaces, and to *invoke* operations in those interfaces.

In the engineering infrastructure, the binding necessary for invocations is provided by a remote procedure call protocol known as REX (Remote EXecution protocol). This is layered on top of a generic transport layer interface known as a *message passing service* (MPS). A number of additional protocols may be included at both the MPS and the execution protocol levels and these may be combined in a number of different configurations. The infrastructure also supports lightweight threads within objects so that multiple concurrent invocations can be dealt with.

All the above engineering functionality is collected into a single library, and an instance of this library is linked with application code to form a *capsule*. Each capsule may implement one or more computational objects. In the UNIX operating system, a capsule corresponds to a single UNIX process. Computational objects always communicate via invocation at the conceptual level but, as may be expected, invocation between objects in the same capsule is actually implemented by straightforward procedure calls rather than by execution protocols. ANSAware currently runs on a variety of operating systems platforms including UNIX, VMS and MS-DOS.

3. Building a Distributed Multimedia Platform

3.1. The Model

Initially it may appear as if the introduction of continuous media [Anderson,90] into the ANSA architecture could be achieved by simply encapsulating multimedia objects behind standard ANSA interfaces, and using high-performance invocation mechanisms to achieve data transmission. On closer examination it becomes clear that this is not the case. Continuous media has fundamentally different characteristics to more traditional static media, and its integration places a number of requirements on the computational model [Coulson,91a]:-

- the need to explicitly represent continuous media transmissions
- the ability to express the synchronisation requirements of continuous media communications
- the ability to express general event-based synchronisation
- the ability to specify and dynamically alter the quality of service (QoS) of continuous transmission communications
- the ability to request sets of resources atomically
- the ability to support group communications, both in terms of invocations and continuous media transmissions

We addressed these requirements by introducing a number of new services into the ANSA architecture. These services are added to the existing ANSAware services (e.g. the trader) to provide new functionality without requiring any changes to the basic ANSA model, that of objects, interfaces, trading and invocation[1]. We call this set of new services the *base service platform.* It consists principally of two types of object; *devices* and *streams*. These are both seen by the higher layers as ANSA services with standard abstract data type interfaces, but they encapsulate the control and transmission of continuous media.

3.1.1. Devices

Devices are an abstraction of physical devices, stored continuous media, or software processes. They may be either sinks, sources or transformers of continuous media data. Most devices present the following pair of interfaces:-

- a *device dependent interface*. The device dependent interface contains operations specific to the device. For example, a camera might have operations such as pan or tilt.
- a generic control or *chain* interface. A piece of continuous media may be visualised as a chain comprising a sequence of *links*, each of which represents an atomic unit particular to the media type in question (for example a frame of video)[2].

The chain interface is shown below.

```
in                              -> [ status ];
out [ rate ]                    -> [ status ];
cancel                          -> [ status ];

start [ numberOfLinks ]         -> [ status ];
stop                            -> [ status ];

seek [ linkNumber ]             -> [ status ];
get_position                    -> [ linkNumber ];
create_endpoint                 -> [ endpoint interface ];
```

[1]The fact that no changes have been made to the basic ANSA model allows us to work within the framework emerging from the current ODP standardisation process.

[2]Chains are an extension of the voice ropes abstraction developed for the Etherphone project [Terry,88], and may be edited using similar operations (e.g. concatenate chains, form a subchain etc.).

This interface is common to all continuous media devices. The *in* and *out* operations selectively set up the device to produce or consume continuous media data (*cancel* is the inverse), whereas *start* and *stop* actually switch the information flow on and off. Sequences of stored media do not have a device dependent interface for control, only a chain interface and hence are often referred to simply as *chains*. A virtual pointer moves through the chain as it is played or recorded, and this pointer may be located and moved using the *get_position* and *seek* operations.

Using the chain interface, clients of a device may create an *endpoint interface* on the device. This interface abstracts over all aspects of a device which are concerned with the transport of continuous media data (essentially it presents a pair of operations, *get_link* and *put_link* through which links can be read or written).

3.1.2. Streams

Streams are the services used to connect devices together via their endpoint interfaces. They are abstractions of continuous media transmissions which map down on to underlying transport protocols.

```
connect_source [ endpoint_group ]          -> [ status ];
disconnect_source [ endpoint_group ]       -> [ status ];
connect_sink [ endpoint_group ]            -> [ status ];
disconnect_sink [ endpoint_group ]         -> [ status ];

prepare [ QoS ]                            -> [ status ];
commit                                     -> [ status ];
uncommit                                   -> [ status ];

get_characteristics                        -> [ charact ];
set_characteristics [ charact ]            -> [ charact, status ];

destroy                                    -> [ status ];
```

The first four operations allow a client of a stream service to connect and disconnect endpoint interfaces. Streams support M:N connections, i.e. they allow M sources to be connected to N sinks. This is modelled by allowing endpoint interfaces to be grouped together, and ensuring streams interconnect endpoint interface groups as shown in figure 2. Endpoint interfaces may be dynamically added to or removed from these groups.

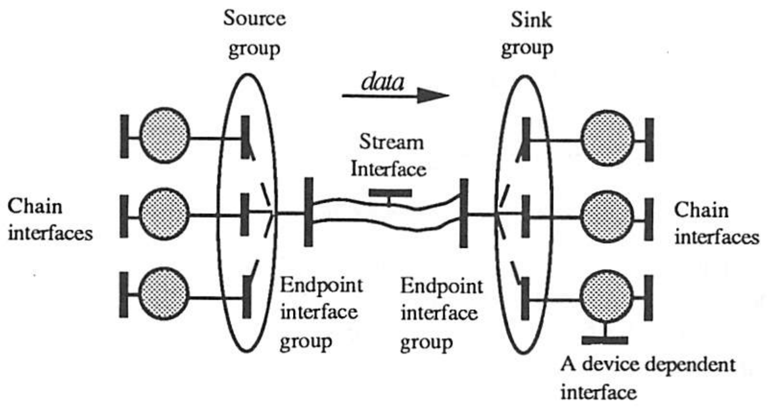


Figure 2: Using streams to connect endpoint interface groups

Groups form an important part of our model, and interfaces may be grouped either to allow the transmission of continuous media data (as shown above), or to allow invocation of a number of interfaces at one time [APM,90], [Coulson,91b].

The model we have presented in this section provides an interface through which application programmers can control continuous media devices and transmission in a distributed environment. Initially all the services described were implemented in the UNIX domain to test the validity of the model. Subsequently those demanding high-performance were also implemented on the MNI, and applications were able to make use of these without modification. Further details of the model described in this section may be found in [Coulson,91a].

3.2. UNIX/Ethernet Implementation

3.2.1. Hardware

The basic hardware configuration in the UNIX domain consists of three multimedia workstations based around Sun Sparc 1+'s. These have been enhanced to provide video and audio capabilities. Video input is provided by a camera and Sun VideoPix card fitted to each workstation. This card is suitable for simple conferencing style applications given its frame display rate of 4 frames per second[3]. Audio i/o is provided using the Sparc's standard audio capabilities.

A video disc player and a CD-ROM drive provide access to stored (read only) sequences of video and audio. Audio and slow scan video are also stored on the standard UNIX file system. The multimedia workstations are interconnected using a standard 10 Mbits/sec Ethernet.

3.2.2. Software

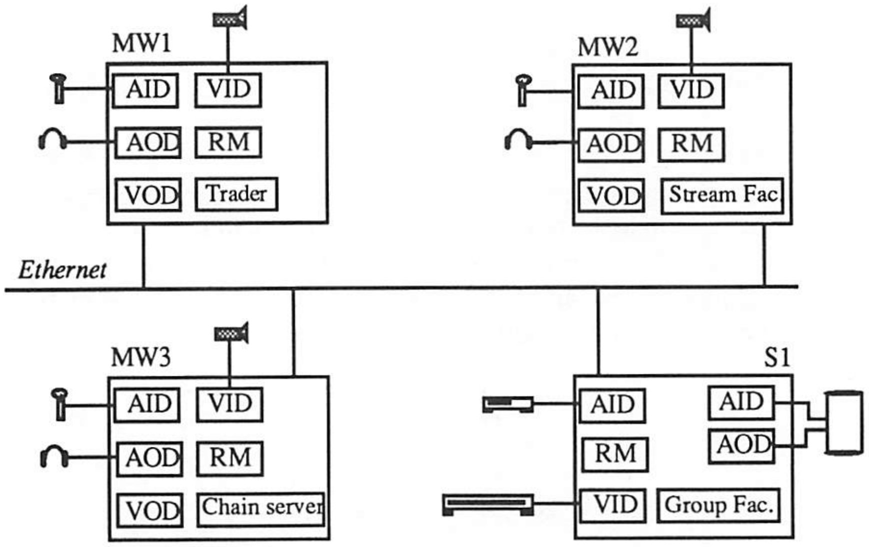


Figure 3 : Key Components in the UNIX/Ethernet Domain

All workstations in our environment run UNIX and version 2.6 of ANSAware. Figure 3 shows the key components in our UNIX/Ethernet implementation. Each box within the workstations (MW1-MW3) or the storage server (S1) represents an ANSA capsule, and is described in the following sections.

[3]Greyscale frames per second on a SPARCstation 1+ (figures from the Sun VideoPix Reference Manual).

Continuous Media Drivers

These are shown as being either audio or video input drivers (AID,VID) or audio or video output drivers (AOD, VOD). Drivers may represent either transient or persistent sources of media. There is a driver corresponding to each continuous media device in our system, and it is to these drivers that streams connect, via the driver's endpoint interface.

The Chain Server

While in practice streams connect to drivers, the existence of drivers is concealed from clients of the base services by the chain server. For each continuous media device, or piece of stored media, the chain server exports a chain interface. It is through these chain interfaces that clients access endpoint interfaces which they can subsequently group and pass to streams. The chain server supports all the chain interfaces within a single capsule, rather than creating a new capsule (and hence UNIX process) for each one. In addition, the chain server maintains state associated with each chain interface such as the current pointer position. The use of a chain server allows drivers to be used by multiple chains, avoiding the need for a separate driver for each piece of stored audio or video.

Group Factory

The group factory allows clients to create empty endpoint interface groups. Once a suitable group has been created endpoint interfaces may join or leave the group. The group factory simply maintains a table of members for each group.

Stream Factory

The stream factory capsule creates new stream objects (again within the same capsule). The groups passed to the stream as parameters to the connect_source or connect_sink operations are interrogated by the stream object to determine their members. The stream object then asks each of these members for their physical address and determines if such a configuration is possible. If it is, then the stream uses either TCP/IP or UDP/IP to form the connection. If it is not, then the operation fails. When endpoint interfaces attempt to join or leave the group, the group consults the stream, and the operation is only allowed to proceed if the stream can support the new configuration.

Resource Managers

A resource manager (RM) resides on each node. This allows capsules which are used infrequently to be checkpointed to backing store. When an invocation for a checkpointed capsule arrives at a node an exception is raised, and the resource manager restores the capsule to its original state, passing on the invocation. More details of this aspect of our work can be found in [Blair,91b].

3.3. Transputer Implementation

3.3.1. Hardware

As previously mentioned, the transputer based MNI unit attaches to a conventional workstation and is responsible for interfacing workstations to a high speed network. In addition to this it transforms the host machine into a multimedia workstation by managing all continuous media sources and sinks at that workstation. The connection to the host machine is via a single transputer link as illustrated in figure 4.

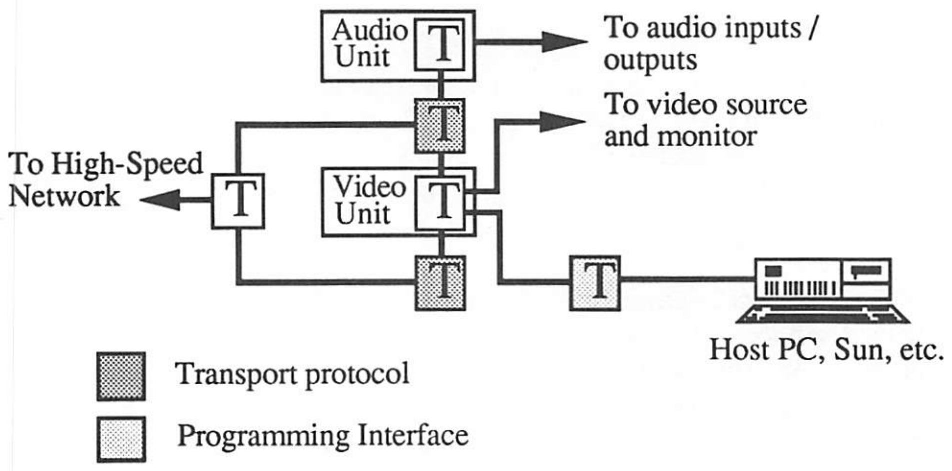


Figure 4 : Multimedia Network Interface Unit

Figure 4 shows the individual components which make up the MNI unit. The video component is built around a Digithurst frame grabber card with a resolution of 720x512 pixels. This drives a separate monitor to the host workstation monitor (the host's monitor need not be connected when the MNI unit is in operation). The audio component is responsible for A/D and D/A audio conversion (currently telephone quality voice is supported, though the MNI may be configured for CD quality audio) and has an attached microphone and loudspeaker. Of the remaining four transputers in the configuration, one is dedicated to providing the programming interface to the unit (see below), another interfaces to the network, and the remaining pair run transport protocol software. All the transputers in the MNI unit communicate via 20 Mhz transputer links.

In our experimental network configuration (see figure 1) we have two PC based multimedia workstations, a Sun 4/UNIX based multimedia workstation and a PC based storage server. The storage server uses three magnetic disks. Two are used as data disks across which continuous media data may be stripped. The third acts as an index for the data disks (more information on the storage server may be found in [Lougher,91]). Instead of a 'real' high speed network, we use a real-time *network emulator* to connect the four stations. This is implemented as a two plane transputer switch with the emulator software running on the network interface transputer of each MNI unit. An emulation is preferred for reasons of flexibility and cost, and also because it is an area of research interest in its own right. We are currently experimenting with emulations of FDDI 1 and DQDB networks [Ball,91].

The Sun workstation acts as a gateway between the transputer and Ethernet environments so that applications can view the two networks as a seamless whole.

3.3.2. Software

The programming interface to the MNI functionality is exactly the same as that in the UNIX environment described above so that client applications can treat the base services platform as a single pool of objects. The programming interface is implemented on the 'root' transputer as shown in figure 4. In particular, this transputer supports a chain server, various device dependent interfaces, and stream and group interfaces and factories. Support of these interfaces has necessitated porting ANSAware to the transputer hardware.

The ANSA capsules running on the root transputer communicate with the processes controlling the continuous media devices by means of a private packet protocol. Most of these drivers are written in OCCAM. For display and video window purposes, we have ported and adapted the X11

R3 window server. The X protocol itself has not been altered as we present video windows as standard continuous media devices with chain, endpoint and device dependent service interfaces.

The two transport transputers run an experimental rate-based transport protocol which is used to carry continuous media in addition to conventional traffic (e.g. ANSA RPC and X11 packets). The protocol supports connections with real-time guarantees in addition to a high degree of QoS configurability. Continuous media traffic is exclusively sourced and sinked within the MNI units themselves. Client applications running on the host workstations are able to access video and audio data generated or stored on the MNI units, but without any real-time guarantees.

The MNI transport protocol is intimately associated with an *orchestrator* process which provides synchronisation between separately stored but semantically related continuous media data streams (e.g. the audio and video components of a film clip). This process is controlled by a manager on the root transputer which exports a 'synchronisation manager' interface to allow control by ANSA client applications. The synchronisation manager also supports an event-based style of synchronisation so that actions can be scheduled on the occurrence of events. For example, an application can arrange to display a text caption when a particular video frame is displayed.

4. Experiences

4.1. Developing Applications with the Base Services Platform

The objective of our research is to build a platform for supporting distributed multimedia applications. To evaluate its success we are developing a number of pilot applications. To be valid, these pilot applications must represent real world scenarios within which multimedia is expected to have a significant impact [Williams,91]. Our approach therefore, has been to involve various end user organisations through which a number of potential application areas have been identified. As an example, after consultation with a major multinational chemical company we decided to develop a microscope controller application based on the work of the company's microscopy research team.

The microscope application provides groups of scientists with remote access to any one of a number of electron or optical microscopes located on a network. Each device when selected will send its video output to a number of user workstations. A view only mode is available whereby each user can view the output of the microscope and control (for example) the size and colour of the video image. If required a record mode can be selected where users can record the microscope output complete with voice annotation. This recording can then be saved into a multimedia document which contains the various media components and a script describing the document structure. Once created the document is available in the system for other applications such as mail tools, document editors etc.

In addition to the microscope application, a number of other test applications have been implemented including an audio/visual telephone and a video jukebox. A limited form of multimedia document is also available along with tools for presenting documents to the user. However, the following sections concentrate on the experiences gained while implementing the microscope application since it encompasses most of the functionality of the other applications.

4.1.1. The application programmers interface

The implementation of the microscope application requires the communication of audio and video between a number of workstations. In addition, both audio and video information must be stored at the user's request. The base services platform provides the necessary support for the communication and storage of information through the stream and chain abstractions described earlier.

The actual implementation of the microscope application is fairly simple in that it requires the acquisition of a number of base services which are then configured accordingly. Considerably more effort would be required to build the application without the support of the platform, particularly since some of the microscope devices are located in the UNIX domain whilst others are attached to the transputer system. The support platform allows these services to be accessed and configured in a standard way irrespective of their particular location.

The application in view only mode is the simplest case and requires a single video stream which connects two endpoint groups. The source group consists of a single source video chain (i.e. the selected microscope device), whilst the sink group consists of a varying number of video window chains depending on the number of users viewing the microscope output.

Record mode has the additional complexity of needing an audio connection, and an extra sink chain to be added to the video stream sink group. In this case there are several source chains connected to the audio stream representing microphones on each user's workstation. The sink chain on the audio stream is a persistent audio chain through which audio can be recorded. The configurations for view only and record mode are summarised in figure 5. The devices are shown as lightly shaded circles with three interfaces: a device dependent interface, a chain interface and an endpoint interface. Persistent sinks of continuous media (chains) are shown as heavily shaded circles which support only chain and endpoint interfaces (see section 3.1.1.).

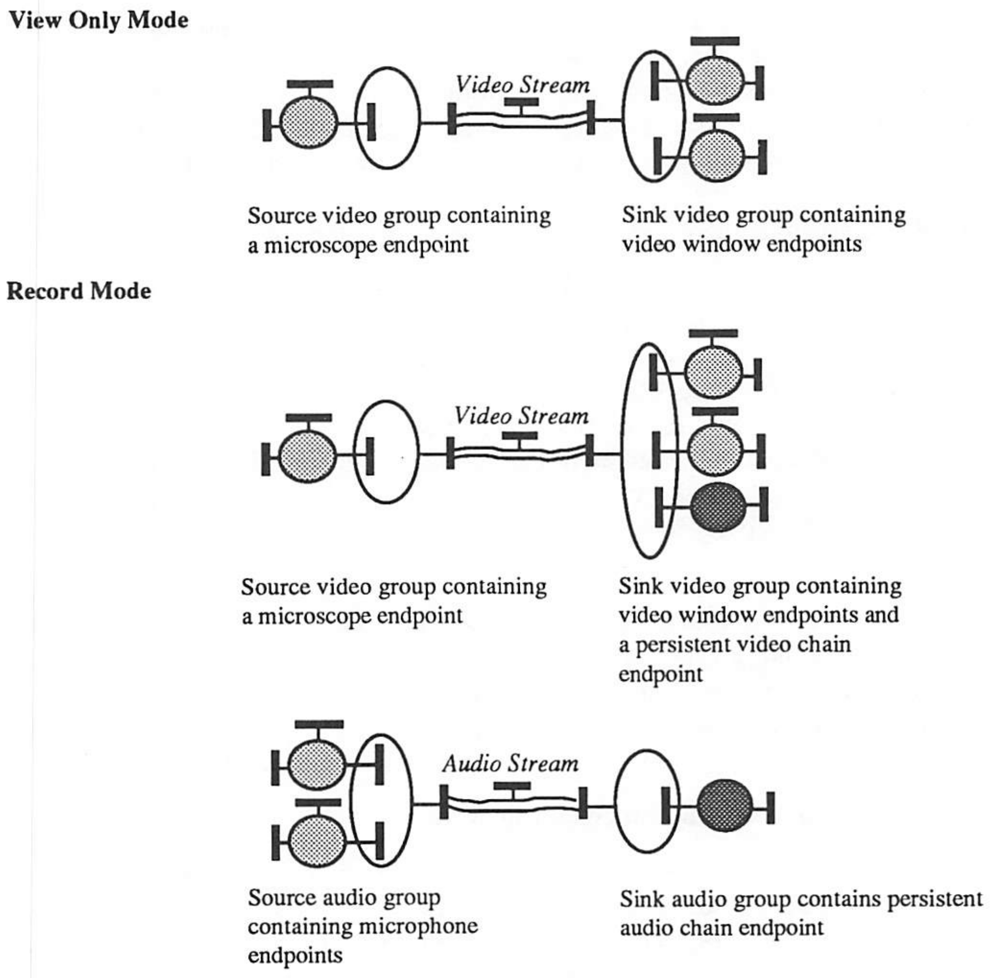

View Only Mode
Video Stream
Source video group containing a microscope endpoint
Sink video group containing video window endpoints
Record Mode
Video Stream
Source video group containing a microscope endpoint
Sink video group containing video window endpoints and a persistent video chain endpoint
Audio Stream
Source audio group containing microphone endpoints
Sink audio group contains persistent audio chain endpoint

Figure 5 : Base Service Configuration to Support the Microscope Application

The description above shows how easy it is for developers to quickly connect together sources and sinks of information for both storage and communication. When actually writing the code however there are still a large number of invocations required to configure the support services. This would seem to indicate that in the general case the level of abstraction is too low. Work is now on-going to provide a *call service* for developers whereby much of the burden of configuring streams, endpoint groups and chains is removed. A call service simply takes a list of source information chains and a list of sink chains and forms the necessary connection. Application programmers who require the full flexibility offered by maintaining access to individual base service components may ask the call service to list those components involved in the connection it has established. This allows programmers to selectively configure individual components, or delegate responsibility to the call service by simply specifying some desired, abstract, quality of service for the connection.

4.1.2. Application performance

The individual performance of the support services for storage, communication and invocation is discussed in section 4.2. However, one of the most interesting aspects of the microscope application is its level of performance in terms of reaction time to user commands. The use of the support platform involves a large number of services which may be distributed throughout the system. These services quickly consume processor resources and reduce the effectiveness of the system. This problem can be noticed with both the loss of audio and video data, and the time taken to complete user commands passed to the application through its X-Windows interface.

Balancing the load on the system by spreading the support services across the available workstations achieves a better level of performance. However, even with the services running on a number of machines the response time is unimpressive. This is mainly due to the large number of invocations on support services required to complete a single user command such as the pressing of the record button. In light of this experience we are investigating possible solutions, as outlined in section 4.2.

4.2. Engineering the Base Services

4.2.1 MNI design

The MNI unit is primarily intended as a testbed for the exploration of design and implementation strategies in multimedia workstation support and network interfacing. Because of this goal, a major requirement of the design was a high degree of flexibility and configurability. In our design we have attempted to fulfil this requirement through the use of transputers as the base hardware component.

In general, we feel that this choice has been vindicated as it has proved easy to reconfigure the unit in both hardware and software terms. Software configurability is made relatively painless by the use of OCCAM and its associated configuration tools, and hardware configurability is simply a matter of altering switches on the transputer cards. As an example of software flexibility, an earlier design dedicated the 'transport' transputers to running software compression processes, and ran both the network and transport layers on the network transputer. We later obtained superior results by discarding compression altogether and giving more resources to the transport layer by splitting it into separate send and receive modules and moving them onto the transport transputers. This change required only a minor modification to a configuration file and a subsequent recompilation. Incidentally, we intend to re-introduce video compression in the near future by replacing the current video card with a newer version incorporating built in JPEG compression.

The major drawback with the transputer approach is the relatively disappointing performance. In particular, we are relying on transputer links for all communication and these have proved to be a serious bottle-neck. Although the links have a theoretical bandwidth of 20Mbits/sec, this is considerably reduced in practice due to an embedded protocol for reliable link transfer. In addition, we use physical link extenders between the workstations and the network emulator to allow a degree of physical distribution, and this further reduces the available link throughput. Another system bottle-neck is the switching speed of the network emulator. However, as the two plane transputer based switch is a relatively isolated and self contained component it is an ideal candidate for replacement by more specialised hardware.

An indication of the performance achieved with the current configuration is given by the following figures. An end to end video stream rate of 12 frames per second with frames of 64x180 and 32 bits/pixel is obtained for a point to point connection. This represents an end to end throughput of about 4.5 Mbits per second. The figure attainable over the bare FDDI network emulator without the transport layer overheads is 11.5 Mbits per second, and the raw throughput of the switch itself (with the link extenders) is 14 Mbits per second. These figures are summarised in table 1 which shows the performance degradation caused by the introduction of the various components.

	Theoretical Max.	Transputer link extenders in place.	Network emulator in place.	End-to-end throughput.
Throughput (Mbits/sec)	20	14	11.5	4.5

Table 1 : Performance figures for the experimental configuration

The significant overhead incurred by the transport protocol is largely explained by our implementation techniques: i.e. the communications protocols are implemented as packet trains passed along a pipeline of processes each of which represents a protocol layer. The OCCAM programming language encourages such modular, process based programs. However, the overhead in message passing is considerable. This is unavoidable when messages are passed across physical links but unfortunate where messages are being passed between processes running on the same transputer. The strength of OCCAM is that it has allowed us to implement naive, rapid prototypes which can later be re-implemented using shared memory communication and/or dedicated hardware to improve performance.

One aspect of the design which we are particularly happy with is the integration of the X-server into the MNI unit. This makes the MNI unit extremely self sufficient: the main role of the host workstation becomes the provision of a supportive environment for application programmers.

The port of the X-server was not entirely straightforward. It required a total re-implementation of the low level TCP/IP based communications so that clients running in the Ethernet domain could transparently access X servers on MNI workstations. Many operating system services assumed by the X-server also had to be re-implemented or simulated in the bare transputer environment. The major change, however, was that the server was split over two processors. Most of the X-server code runs on the root transputer but the low level graphics routines run on the video-card transputer. This was implemented within the X-server framework by providing an emulation of shared memory between the two processors running over the transputer link. Because the graphics routines used by the X-server run on a dedicated processor, the server itself is extremely fast. However, the bottle-neck is in the X protocol path between clients and the server: this is the host/MNI link which suffers from extremely poor latency characteristics (although it performs well in terms of sustained throughput). This latency problem is also reflected in the UNIX/transputer RPC performance discussed in section 4.2.3. In the near future we intend to modify the hardware configuration to incorporate *two* root transputers, each with its own link to the host workstation. This will allow X and the ANSA base services to run on separate transputers, halving the load on each link.

The X-server was also altered to incorporate video windows. Rather than provide application access to video windows through an extension of the X protocol, we present video windows to the application programmer as *devices* in the same way as any other continuous media source or sink. Video windows are fully integrated into the X display in terms of clipping and exposures, but control operations such as resizing or moving are effected via an ANSA interface. We feel that this abstraction is superior to the alternative choice of presenting video windows as augmented versions of ordinary X windows, and thus drawing an unnatural distinction between video windows and other continuous media sources/sinks.

To sum up, the medium to long term solution to most of the problems identified in this section is to move to dedicated hardware or at least dual port shared memory between transputers. The complexity inherent in these solutions varies considerably: it would be relatively trivial to replace the two plane transputer switch in the network emulator, whereas a hardware implementation of the transport and orchestration modules would be a significant undertaking. However, we feel much more confident about taking such steps having had the chance to iterate towards a stable design in the transputer environment. Finally, notwithstanding the current performance limitations, the prototype MNI unit is a usable multimedia workstation testbed, and the network emulator has proved its worth as a basis for experimental continuous media transport protocols and synchronisation experiments.

4.2.2. Multimedia in the UNIX/Ethernet domain

The UNIX/Ethernet domain implementations of time critical services proved surprisingly good at manipulating continuous media. An implementation of a simple audiophone application was able to support three users (six voice channels) comfortably. We were also able to provide demonstration applications which incorporated slow scan video being transmitted across the Ethernet.

Whilst the transmission of continuous media types could be supported we found that presenting these media types at their destination was more problematic. For audio the problem arose from the need to mix multiple incoming audio streams for presentation via a single loudspeaker in software. This led to a considerable reduction in the quality of service obtained as the number of incoming streams increased. To overcome this problem the audio drivers were re-written to be optimised for the general case, i.e. data being received on only a single channel regardless of the number of channels connected. This was achieved by ignoring channels unless they contained signals above a certain threshold. When the threshold was exceeded on more than one channel the driver attempted to mix these channels but the result was often poor. As we had expected, for an application such as the audiophone the solution was adequate. For the majority of the time only a single person talks at any give time. Two or more people talking at the same time is normally regarded as a 'collision', and the users themselves back off with 'after you' style comments. In this case the information lost as a result of the collision is normally repeated. Clearly such a solution would not work for applications which required the simultaneous presentation of multiple audio channels, in which case mixing would probably need to be performed in hardware.

Presentation of multiple video images at a single destination also caused problems, particularly with the X colour map. X applications maintain their own colour map which is installed whenever the user interacts with the application (by moving the cursor into the applications window). However, it is usual in multimedia groupware applications to be working within one window (e.g. an editor) whilst wishing to view a number of other windows (e.g. video images of co-workers). As only a single colour map may be installed at any given time, either the editor or the video images will be incorrect. We have not as yet implemented a solution to overcome this difficulty.

Despite these problems, by far the greatest difficulty encountered was with the *control* of continuous media. As outlined in section 4.1 relatively simple operations such as recording a piece of audio take a large number of invocations to achieve even once the configuration has been established. We observed that as the quantity of continuous media traffic on the network increased, so the likelihood of control operations (such as record) failing increased. This is clearly unacceptable. Whilst a degradation in the QoS of the continuous media streams might be acceptable, the loss of controlling invocations is not. We can therefore identify a clear need to be able to reserve resources in advance, not only to support continuous media streams but also to enable guaranteed performance of control invocations. Indeed, if invocations could be guaranteed to complete - or at least arrive at their destination - within certain time bounds then they could be used to synchronise continuous media streams. To achieve such invocation guarantees in UNIX is obviously not possible. We are therefore investigating ANSAware in conjunction with the Chorus [Herrmann,88] operating system which provides comprehensive real-time programming facilities as a suitable platform for future applications development.

4.2.3. Using ANSAware

As a result of our use of ANSAware we are able to evaluate it according to two separate criteria. Firstly, as a platform for building distributed applications, and secondly on the grounds of its extensibility and portability.

In the first case, as a platform for building distributed applications we have been pleased with the overall performance of ANSAware. The need to know the types that an object will interact with in advance (in order that the appropriate stub libraries may be compiled with the object) placed certain limitations on application design. However most of these limitations can be overcome by the judicious use of 'generic' interfaces supported by multiple objects.

ANSAware also leaves the application designer to determine the appropriate mapping between objects and capsules. Capsules (single UNIX processes) can support any number of objects. As inter-capsule object invocation is optimised at compile time to produce local procedure calls, placing multiple interacting objects within a single capsule can lead to a considerable improvement in performance. However, this increase must be weighed against a corresponding decrease in flexibility. We experimented with different object to capsule mappings. For example, figure 3 shows the stream factory and the group factory as separate capsules, and the the accompanying text describes the interactions which take place between a stream and a group when a connection is established. Whilst this configuration worked, and was ideally suited to our developmental environment, we have since implemented an optimised version with both the group and stream factories in a single capsule. The optimised capsule performs connections at least an order of magnitude faster than its distributed counterpart.

Incorporating multiple objects into a single capsule can lead to problems other than those associated with a loss of flexibility. Capsules which receive multiple invocations from different clients exhibit a marked reduction in performance. This led us to attempt to minimise the number of invocations capsules may be subjected to at a given point in time. This is clearly another factor which which must be accounted for when considering object to capsule mappings.

Object invocation is also affected by our maintaining two domains, a UNIX domain and the MNI domain. The additional code required to achieve the routing between these domains, and to support our persistence mechanism lead to an increase of around 10% in the overall time taken to perform object invocation (where both objects reside in a single domain). However, this extra delay is relatively small when compared to that incurred when invocations traverse the domains (i.e. one of the capsules concerned is in the UNIX domain, the other in the MNI domain) via the transputer link. In this case there is approaching an order-of-magnitude increase in the time taken to perform invocations. As mentioned in section 4.2.1. this is due in part to the implementation in the UNIX domain of the link as a UNIX device driver. As such the link is optimised for high throughput, rather than the low latency desirable when supporting bursty traffic such as object invocations. We are currently investigating possible solutions to this problem (section 4.2.1.).

From an engineering perspective ANSAware proved relatively easy to both port and extend. Operating system dependencies are isolated within a few key libraries. Once ported it provides a suitable platform for developing interworking applications as demonstrated by the ease with which objects within our two domains may communicate.

5. Related Work

This section briefly reviews related work in the field in an attempt to place our work in context. We cover Bellcore's Touring Machine [Arango,90], the Pandora multimedia hardware extension [Hopper,90] from Olivetti Research, Cambridge, UK, and the ACME (Abstractions for Continuous Media) Server [Anderson,91] from the University of California at Berkeley.

The Touring Machine was conceived as an object-based platform for distributed multimedia applications, and thus has very similar aims to our work. It has been used as the basis for a number of applications which have been successfully used in field situations over long periods of time (e.g. Cruiser [Fish,89]). The Touring Machine platform offers lower level facilities than our base services: in particular features such as trading and object invocation with automatically generated marshalling stubs are not provided. Also, the Touring Machine is based on analog networking for video and audio which reduces flexibility. For example it is only possible to display four video windows at a time, as the screen is statically partitioned into quadrants.

There is a difference in emphasis between our 'open systems' philosophy and Bellcore's telecommunications perspective; the Touring Machine is deliberately partitioned between the domains of the network and its customers for administrative purposes. These differences are documented in a joint study undertaken by Bellcore and ourselves in [Ruston,91]. At present access to all parts of our base services by applications is unconstrained and we have no support for accounting or other administrative tasks. Whilst we recognise the importance of providing such facilities in our platform we have not as yet addressed this area in any detail.

Olivetti's Pandora's Box is a hardware extension board which plays a similar role to our MNI unit. Both systems handle all audio/video streams at their host workstation and are based on transputer technology. As with the Touring Machine, Pandora is installed on a large number of stations in an everyday working environment, and supports a wide range of applications including video conferencing, video mail and workstation access to TV and radio servers. It is a fully digital system and is supported by the Cambridge Fast Ring (CFR) network. The Pandora unit supports only monochrome video in small, fixed window sizes, but is able to simultaneously handle a much larger number of video and audio streams than our MNI unit with its 24 bit colour. The relatively high performance is due to the use of shared memory for inter-transputer communication, and the use of a faster network.

Although there are close analogies between Pandora and the MNI unit, our primary aim has been to provide a general purpose support platform for distributed multimedia applications. This aim has been validated by the use of the same platform over the two very different support infrastructures as described in this paper. The emphasis of Pandora, on the other hand, is very much on the hardware. The programming interface is low level with no location transparency, and the programming of devices is ad-hoc and device specific.

Berkeley's ACME server is a UNIX process which handles all continuous media sources and sinks at a single workstation. No special hardware is required; the system is Ethernet based and TCP/IP is used for continuous media communications. The philosophy of the Berkeley work is that continuous media processing should be fully integrated into the operating systems environment rather than delegated to add-in boards which do not give application threads direct access to real-time continuous media information. Although it is not yet possible to match the performance of dedicated hardware solutions, Berkeley believe that this situation will soon change and that application programmers will demand the integrated model. As a result of this philosophy, the emphasis of the Berkeley work is on modelling resource allocation and the efficient scheduling of user-level threads to meet continuous media deadlines.

A second difference in emphasis between ACME and our base services is that the point of departure for the ACME architecture is existing windowing systems such as X. Thus ACME can be viewed as a conventional window server with extensions for video windows and audio support. Our approach, on the other hand, is to unify the architecture around the notion of the continuous media source/sink. Thus video windows are modelled in an analogous fashion to loudspeakers, with a generic continuous media related chain interface, rather than being closely related to conventional X-windows.

6. Concluding Remarks

The experiences gained from implementing and using a distributed multimedia platform are proving invaluable as we consider our future plans. Our first design aimed to provide maximum flexibility in all aspects of the architecture. The components necessary for manipulating continuous media were placed within a single enhancement unit so that it could be connected to any workstation. The hardware consisted of a set of easily re-configurable transputers, and the software was implemented in a modular, easily re-usable fashion using OCCAM. Within the UNIX domain software was written entirely as ANSA capsules, and new services could be introduced without disrupting the existing infrastructure. Flexibility was also provided at the application programmers interface, with services appearing as a uniform pool of objects.

However, supporting flexibility in the platform has inevitably led to performance overheads. It is these overheads which we aim to reduce in subsequent implementations of our services. To this end, we are, as outlined above, investigating two avenues of future work. Firstly, we intend to replace the transputers in the MNI unit with dedicated hardware. This is possible since we now have a clear idea of the interface we require to the MNI unit (having been validated through the implementation of real world applications). Secondly, we are considering moving from UNIX to Chorus as a support platform in the workstation environment. We anticipate that this would not only lead to a considerable increase in performance, but would also allow us to experiment with real-time resource allocation and object invocation.

The application programmers interface will remain relatively unchanged. We have been particularly pleased with this approach which has allowed us to add and modify servers with the minimum of disruption to application programmers. Whilst for some purposes the interface is at too low a level, our experiments have shown how higher level services may be successfully built upon the base services platform.

Acknowledgments

Many people have contributed to the development of the Lancaster base services platform. The authors would particularly like to acknowledge the work of Frank Ball, Frankie Garcia, Phillip Lougher and Andrew Scott on the MNI. This work was carried out as part of the MNI project (funded under the SERC Specially Promoted Programme in Integrated Multi-Service Communication Networks and co-sponsored by British Telecom Labs) and the Zenith Distributed Object Management project (funded by the Information Engineering Directorate of the DTI/SERC under grant GR/F 37627).

7. References

[Anderson,90] D.P. Anderson, S.Y. Tzou, R. Wahbe, R. Govindan and M. Andrews, "Support for Continuous Media in the DASH System", *Proc. of the 10th International Conference on Distributed Computing Systems,* Paris, May 1990.

[Anderson,91] Anderson, D.P., R. Govindan, and G. Homsy. "Abstractions for Continuous Media in a Network Window System." *Proc. of the International Conference on Multimedia Systems,* Singapore, January 1991.

[APM,89] APM Ltd. "The ANSA Reference Manual Release 01.00", Architecture Projects Management Ltd., UK, March 1989.

[APM,90] APM Ltd. "A Model for Interface Groups", Architecture Projects Management Ltd., UK, 1990.

[Arango,90] Arango, M., and R. Clayton. "An Infrastructure to Support Multimedia Applications", *Internal Report,* Bellcore. 1990.

[Ball,90] Ball, F., D. Hutchison, A. Scott, and D. Shepherd. "A Multimedia Network Interface (MNI)." *Proc. of the 3rd IEEE COMSOC International Multimedia Workshop (Multimedia '90),* Bordeaux, France, 1990.

[Ball,91] Ball, F., and D. Hutchison. "Performance Evaluation of FDDI by Emulation." *Proc. of the 7th UK Computer and Telecommunications Performance Engineering Workshop,* Edinburgh, UK, July 1991, (Springer-Verlag), pp. 179-184.

[Blair,91a] Blair, G.S., G. Coulson, N. Davies, and N. Williams. "Incorporating Multimedia into Distributed Open Systems." *Proc. of EurOpen'91,* Tromsø, Norway, 1991.

[Blair,91b] Blair, G.S., and N. Davies. "Incorporating Multimedia in Distributed Object-Oriented Systems: The Importance of Flexible Management." *Proc. of the International Workshop on Object-Orientation in Operating Systems (I-WOOOS'91),* Palo Alto, Calafornia, October 1991.

[Coulson,91a] Coulson, G., G.S. Blair, N. Davies, and N. Williams. "Extensions to ANSA for Multimedia Computing", *Internal Report ref. MPG-90-11, and submitted for publication,* Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, June 1991.

[Coulson,91b] Coulson, G., J. Smalley, and G.S. Blair. "Implementation of a Group Invocation Architecture in ANSA", *Internal Report*, Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, August 1991.

[Fish,89] Fish, R.S. "Cruiser: A Multi-media System for Social Browsing" *The ACM SIGGRAPH Video Review Supplement to Computer Graphics,* Vol.: 45, No.: 6, Videotape, 1989.

[Herrmann,88] Herrmann, F., F. Armand, M. Rozier, M. Gien, V. Abrossimov, I. Boule, M. Guillemont, P. Leonard, S. Langlois, and W. Neuhauser. "CHORUS, A New Technology for Building UNIX Systems." *Proc. EUUG Autumn Conference,* Cascais, Portugal, October 3-7 1988, pp 1-18.

[Hopper,90] Hopper, A. "Pandora - An Experimental System for Multimedia Applications." *Operating Systems Review SIGOPS* Vol: 24 No.: 2, April 1990, Pages: 19-34.

[Lougher,91] Lougher, P., and D. Shepherd. "A Multimedia Storage Server", *Internal Report,* Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, November 1991.

[Ruston,91] Ruston, L., G. Coulson, N. Davies, and G.S. Blair. "Integrating Computing Standards and the Telecommunications Industry: A Tale of Two Architectures", *Proc. of the 2nd International Workshop on Network and Operating System Support for Digital Audio and Video,* Heidelberg, Germany, November 18-19 1991.

[Terry,88] Terry, D.B., and D.C. Swinehart. "Managing Stored Voice in the Etherphone System." *ACM Transactions on Computer Systems Vol. 6 No. 1,* February 1988.

[Williams,91] N. Williams, G.S. Blair and R.A. Head "Multimedia Computing: Applying the Technology", *Internal Report ref. MPG-91-10,* Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK, April 1991.

A Universal Distributed Programming Paradigm for Multiple Operating Systems

David L. Cohn, Michael R. Casey, Paul M. Greenawalt, John E. Saldanha
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana 46556
Contact: dlc@cse.nd.edu

ABSTRACT

This paper describes the addition of a distributed programming paradigm to three substantially different operating systems. The paradigm is based on the data unit abstraction, which was originally developed as an element of ARCADE, a research-oriented distributed kernel. It has been added to VM/CMS, OS/2 and Mach and provides transparent cooperation between processes in all three operating systems. Data units are structured distributed shared memory elements which allow implicit communication and translation. They are a high-level abstraction, and allow seamless cooperation even between machines of different architectures. Data units are directly accessible and do not interfere with local operating system services. Also, applications which use only data units as a cooperation mechanism are portable between operating systems. Thus, data units can be viewed as a universal cooperation mechanism.

1. Introduction

Distributed programming implies close cooperation between autonomous elements of execution running on separate machines. Numerous paradigms for this cooperation have been proposed and implemented in various operating systems [1][2][3][4]. In general, it is clear that successful paradigms should be both easy to use and efficient. However, there is no general agreement on which paradigm best achieves these two seemingly contradictory goals. This paper proposes the use of the *data unit* paradigm as a universal cooperation mechanism. It shows how data units have been added to three existing operating systems to allow truly heterogeneous interoperation.

The ARCADE architecture [5][6] introduced the data unit paradigm and the related *data unit link*. Implementations of ARCADE have shown that these are effective and efficient mechanisms for constructing distributed applications. They are easy to use since they allow applications to be built naturally, without explicit communication. Also, data units allow direct use of physically shared memory when it is available and are competitive with other forms of remote cooperation when communication is necessary. Finally, because data units allow implicit data movement between parts of an application, they are well-suited to asynchronous algorithms [7]. By running asychronously, algorithms can minimize the impact of unavoidable communication delays [8].

The ARCADE architecture was developed as a complete, but minimal, set of abstractions and services for a distributed operating system kernel. It defines the active *task* abstraction as well as the passive *data unit* abstraction. An ARCADE task is similar to most kernel-level task definitions. It differs somewhat in that its address space contains only data units and it can control other tasks with *input/output lines*. While tasks are essential to making ARCADE a complete kernel, data units and data unit links are its most attractive abstractions.

Data units offer important benefits for distributed programming. They are structured carriers of code or data. An ARCADE task creates a data unit much like a conventional process allocates memory. However, rather than specifying the *size* to be allocated, the task specifies the *structure*. This information is permanently associated with the data unit, and facilitates translation when the data is moved.

Data units allow the definition of a global data object identification mechanism called *data unit links*. These are pointer-like variables whose value is managed by the system. They can be transferred between machines, can link data units on multiple machines and remain valid even if their target moves.

ARCADE data units can be *moved* or *shared*. Moving is essentially message passing with implicit translation. It can be used as the basis of well-known cooperation mechanisms such as remote procedure calls. Sharing allows dynamic creation of either physical or distributed shared memory [9]. Normal coherency mechanisms for distributed shared memory [10][11][12] can be used for data units. Also, *data unit locks* permit application controlled coherency and avoid the *stale data problem* [13]. Experience has shown that application controlled coherency is particularly effective [14]. Indeed, it is the flexibility of sharing data units which makes the paradigm unique and powerful.

Data units are an attractive candidate for a universal cooperation mechanism. They allow processes to cooperate by simply manipulating locks and by writing to memory. However, ARCADE is a cohesive architecture and several of its task-related concepts are important for data units. For example, all tasks have globally unique, location transparent names. A destination task's name is used by a sender when a data unit is transferred. Thus, to make data units universal, it was necessary to identify a workable subset of the ARCADE architecture which would be small enough to be easily ported and large enough to support data units.

In order to demonstrate that data units could function as a universal distributed programming paradigm, they were added to three widely available operating systems. The necessary service set was added to VM/CMS [15], OS/2 [16] and Mach [17][18]. The latter two operating systems function on 386-based computers, which also support ARCADE. However, VM/CMS is an IBM System/370 operating system so the implicit translation of data within data units is exercised.

Data units are built at the heart of the ARCADE kernel. However, it seemed impractical to implement them at the kernel-level of the other operating systems. This would have required an intimate familiarity with complex and potentially unobtainable source code. Therefore, two different approaches to adding data units were tried: a kernel extension and user-level services. The code for each of these implementations was kept as close to ARCADE as possible.

This introduction is followed by four sections. Section 2 describes the subset of ARCADE services that are essential for data units. The following section details how data units were added to the three other operating systems. Experience in using these data unit implementations are dealt with in Section 4. The final section discusses lessons learned, conclusions and future work.

2. Services related to data units

Data unit services can be grouped into four types: distributed data services, link and lock services, naming services and task creation services. The distributed data services are those which define and manipulate data units. Links and locks are the extended services for data units that give data units their flexibility and power. Naming is used so that data units can be transferred, and task creation allows construction and operation of distributed applications. Each of these service sets is described below.

Table 1 - Distributed data services

`allocate()` - create a data unit
`release()` - remove data unit from address space
`qstruc()`, `qsize()` - access data unit information
`move()`, `share()` - transfer data unit
`receive()` - add notification packet to address space

The distributed data services are listed in Table 1. The **`allocate()`** service is used to create a data unit and map it into the caller's address space. The caller gives the address of a data unit structure specification and the address of the data unit is returned. A data unit is essentially discarded with **`release()`**. If the data unit happens to be shared, it is removed from the caller's address space, but it is not actually deleted. Information about data units can be obtained with **`qstruc()`** which returns the structure and **`qsize()`** which gives the size.

The next four services relate to data unit transfers. Both **`move()`** and **`share()`** allow the caller to specify a destination task. With **`move()`**, the data unit is removed from the caller's address space, but with **`share()`**, it is not. In either case, a *notification packet data unit* is sent to the destination. The notification packet contains information about the data unit including its origin, size and a data unit link pointing to it. The destination task uses the **`receive()`** service to transfer the notification packet to its address space.

In order to access the data unit itself, the receiver must use **`access()`**, shown in Table 2. This service has a data unit link as an argument, and it causes the target data unit to be mapped into the caller's address space. Once accessed, a data unit

Table 2 - Link and Lock Services

`access()`	map dul target into address space
`wait_du()`	high-level wait, receive, access
`setlink()`	assign data unit link
`copylink()`	duplicate data unit link
`lockdu()`	lock data unit
`unlockdu()`	unlock data unit

appears as directly accessibly memory. Actually, **`access()`** is a general purpose data unit link de-referencing service, and it works with any data unit link.

The **`wait_du()`** service allows a task to wait until a data unit arrives, receive the notification packet and access the data unit itself. Ideally, a waiting task would be blocked. In ARCADE, input lines are used to make the task sleep until a data unit arrives, but other implementations may allow it to spin. As in ARCADE, **`wait_du()`** is normally implemented as a library routine which calls **`receive()`** and **`access()`**.

In addition to **`access()`**, there are two other services for data unit links. With **`setlink()`**, the caller specifies a data unit link and an address within a data unit. The service causes the data unit link to point to the indicated data unit. The **`copylink()`** service is a shorthand way to duplicate access to a data unit. Data unit links have been found to be an effective way to construct dynamic data structures which can span machine boundaries. The entire structure can be stored in an innovative file system [19].

The **`lockdu()`** and **`unlockdu()`** services are used to control the consistency and coherency of a shared data unit. Consistency is maintained with database-like tools. A task asserts a *read lock* to assure that data does not change while it is reading or a *write lock* to prevent readers from seeing partial writes. When a write lock is released, changed data is propagated to data unit replicas. Thus, a task can control coherency by judicious use of write locks.

Table 3 - Naming Services

`enroll()`, `unenroll()`	create, remove task names
`getuid()`, `getlname()`	translate names and uids
`machines()`	return list of available machines

While these services are sufficient to define data units, they are not complete enough to make data units useful. For example, **`move()`** and **`share()`** require specification of the destination task. In ARCADE, all tasks are given *names* when they are created. The naming services listed in Table 3 allow processes in other operating systems to acquire and manipulate ARCADE-like names. A process that wishes to use data unit services calls **`enroll()`** to identify itself and establish its name. The full name, or *logical name*, consists of the machine name followed by a locally unique task name. A task can release its name with **`unenroll()`**.

Since logical names can be variable length, a fixed length *unique identifier*, or uid, is usually used to refer to processes. It is this identifier that is used in **`move()`** and **`share()`**. The services **`getuid()`** and **`getlname()`** translate logical names to uids and vice versa. The **`machines()`** service returns a list of all machines that are currently participating in an interconnection.

Strictly speaking, the services in the final class, listed in Table 4, are not needed

Table 4 - Task Creation Services

`namechld()`	- name creation for child task
`maketask()`	- local task creation
`run_pgm()`	- high-level (remote) program start

for data units. However, without them, there would not be a consistent way to create the distributed programs which use data units. The first two services follow the ARCADE model and are the two steps in creating a process. Children must be reliably associated with their parent so the parent declares its intent to have a child. Also, processes can spawn remote children only when they are authorized to do so. Thus, new processes on a given machine are created only by existing processes on that machine. This complicates spawning a remote child and has lead to the two-step process.

Unlike real life, the first step for an expectant parent is to declare the name of its unborn child with **`namechld()`**. The service reserves the name and returns a secret verification *key*. For local child creation, the expectant parent uses this key and the child's name in **`maketask()`**. It also specifies the address of a *program data unit*, a machine-specific collection of the information needed to start a process. For example, it might include a file name and an argument vector or perhaps a pointer to preloaded code.

For remote task creation, the parent asks an agent on the remote machine to call **`maketask()`**. Thus, the parent need not know the format of a program data unit on the remote machine, nor need it send any code. It just transfers the child's name and key together with an indication of where to find the code.

A high-level call, **`run_pgm()`**, combines the two steps and hides the need for an agent. Its arguments include the child's name, the name of a file which contains the code and the name of the machine where the task is to run. In a more complex implementation, **`run_pgm()`** could select the machine based on a load balancing policy.

Taken together, these services constitute the full data unit paradigm. The challenge is to add them in a consistent way to significantly different operating systems. In each case, the addition must have a minimal impact on the operating system environment with implementation details hidden in libraries. Therefore, not only can applications which use data units cooperate *across* operating systems, but they are easily ported *between* them.

3. Implementations of Data Units

The service sets described above have been added to three operating systems to demonstrate the value of data units as a universal cooperation mechanism. In each case, the clients of the data unit services were to be standard processes or tasks as defined by the base operating system. The operating systems were to be unmodified and clients were to use standard tools, through library routines, to access data unit services. This section will describe the three resulting implementations. First, the elements of all implementations will be described.

In every case, there is an entity referred to as the *service provider*. The structure of the service provider varies between implementations. In every case there is a mechanism for the client task or process to communicate with it. Communication is initiated by the client whenever a service is requested. The service provider fulfills the request and returns a result code.

The service provider is responsible for maintaining several data structures. It keeps track of names of tasks and their children and knows the state of data unit locks. It maintains data unit descriptors which contain the data unit structure and the target of each data unit link. Finally, it knows where tasks and data units are, or at least how to find them. Ideally, all of this information should be protected from the clients.

In order to maintain data unit coherency, the service provider has to be able to change things in the clients' address spaces. Whenever a write lock is released, all replicas of a data unit have to be updated. Since they will be in clients' address spaces, the service provider must either write to that space or have some other way of changing it. In any case, the updating must be transparent to the clients.

Clearly, the service provider must be able to communicate with other service providers. ARCADE introduced the notion of an *interconnection* which is a collection of uniquely named machines which can cooperate. The service provider on any such machine has to communicate with the service providers on all other machines. To achieve universality, the nature of this communication has to be independent of the host operating system. Therefore, what goes out "over the wire" is the same for each implementation and for ARCADE itself.

In every case, the service provider requests an available UDP port for transmission and binds to a well-known UPD port for reception. When it first comes alive, the service provider broadcasts a "hear I am" message to the well-known port on the other machines. Current implementations limit the interconnection to one local network, so all active service providers learn the name and address of the new machine.

For best efficiency, the service provider should have a means of temporarily blocking a client and then restarting it. Some services require communication, and it would be wasteful for the client to spin. Finally, the service provider should be able to act as the "agent" which creates a local child for a remote parent. This would allow a uniform implementation of the **`run_pgm()`** service.

It should be noted that each of the target operating systems provides some services for distributed programming. All could be used in remote procedure call systems and all provide standard communication services. Each one also has other unique services for distributed applications. However, these tend to depend on operating system specific concepts and generally are not easily ported. The simplicity of the data unit paradigm makes it attractive as a universal cooperation approach.

3.1 Service Provider for VM/CMS

The VM/CMS implementation is actually an attempt to implement the *entire* ARCADE service set in a virtual machine. Provisions have been made for input/output lines and the associated controls. However, the techniques used would not have changed if only the more limited set described in Section 2 were to be realized.

The VM operating system actually has two distinct elements: CP, the *Control Program*, which creates a set of *virtual machines;* and CMS, the *Conversational Monitor System*, which is a single-user, single-threaded operating system that runs in a virtual machine. The data unit service provider for VM/CMS is implemented using the CMS *nucleus extension facility*. A nucleus extension is memory resident code that can be called by a user program and that can handle interrupts. Data unit services were added to VM/CMS by constructing a nucleus extension to provide the services.

The data unit concept of a *machine* was mapped to a virtual machine. Since CMS is single-tasking, there can only be one task on each machine. Multiple tasks could be accommodated with multiple virtual machines. A task name and UID are assigned when the nucleus extension is loaded. Thus, there is no **`enroll()`** service and any CMS program that invokes a data unit service uses this task name and UID. Currently, a CMS task may not have local children, but it may spawn remote children.

Since the service provider is resident code on the virtual machine, it has unrestricted access to task memory and can use CMS's memory management services. When necessary, it requests that CMS allocate or deallocate memory for data units and internal storage. It only has to remember the locations of the data units.

In this prototype implementation, communication with other service providers is performed via exclusive use of the token-ring hardware. Rather than using a full UPD/IP implementation, UDP compatible frames are transferred directly to and from the hardware. The service provider installs its own interrupt handlers and masks and unmasks interrupts when necessary. The interrupt handler is notified when incoming frames are received and when a send is completed. If, for example, a received frame contains a data unit update, the service provider can simply write the new data to memory.

The nucleus extension approach has some advantages and some drawbacks. It does make task blocking simple. However, it complicates local task generation. Future implementations will use multiple virtual machines and start new tasks on them. A more severe problem is that CMS operates in a flat, unprotected address space. An errant program could write off the end of one data unit and into another, or it could also access the "inaccessible" information stored by the service provider.

3.2 Service Provider for OS/2

OS/2, unlike VM/CMS, does not easily support kernel extensions. Therefore, data unit services are provided by a *data unit services task* (DUST) running at the user level. Standard OS/2 tasks act as clients and interact with DUST through normal user-level interprocess communication. Data units are physically shared between DUST and the client task. DUST communicates with other data unit service providers using standard UDP/IP.

DUST is multi-threaded. Its primary thread dynamically creates additional threads to serve each client task. Other threads allow DUST to concurrently receive remote requests and manage distributed shared memory updates while servicing local clients. Thus, the code is relatively clean, with critical sections to maintain data integrity.

Details of the communication between clients and DUST is buried inside the data unit services library. Local named pipes are currently used because they provide blocking, full-duplex, point-to-point messaging and are, therefore, fairly efficient. They add speed by directly routing responses to the requester. When a client opens a pipe, DUST allocates a thread to service it. When the client has finished using data services, it closes the pipe, and the corresponding thread terminates.

A shared memory facility, such as that in OS/2, is critical to implementing data unit services as a separate task. An OS/2 task's memory is composed of *segments* which contain its code and data. Each data unit, therefore, is implemented as a separate segment in a task's address space. In OS/2, a segment can be declared *shareable* and then shared between tasks.

Consider how DUST uses a sharable segment to respond to an **allocate()** call. First, it allocates a sharable segment in its own address space which corresponds to the size of the requested data unit. It then grants the client task access to the segment. Later, DUST can seamlessly update the data unit by simply writing to the segment in its own memory. When multiple local tasks share the same data unit, DUST can grant each task access to the same physical copy. When the data unit is replicated remotely, DUST cooperates with other service providers via UDP/IP to maintain the illusion of distributed shared memory.

DUST implements both **namechld()** and **maketask()**. For **namechld()**, it enters the name, uid and secret key on a data structure. If the child is spawned locally, the parent sends an architecture-specific program data unit (PDU) and the key along with the **maketask()** call. For OS/2, the PDU contains the name of a local, executable file. It optionally contains command line parameters and an environment segment. By default, all tasks are started as background tasks. The PDU indicates whether the task needs a screen and keyboard and, if so, the task is started as a new session.

The **run_pgm()** service is a library routine. Regardless of where the child is to start, it calls **namechld()**. For a local child, it then builds the PDU and calls **maketask()**. For a remote child, it sends a data unit with appropriate information to an agent on another machine which, in turn, calls **maketask()** there.

3.3 Service Provider for Mach

Data unit services in Mach, like OS/2, are provided by a data unit server task. Clients issue service requests in the form of RPC messages to DUST. The implementation of a data unit service provider in Mach is the youngest of the efforts to port data units to other operating systems. However, experiences with OS/2 and several of Mach's research-oriented features helped the project proceed rapidly. Task creation services for Mach have not been implemented yet.

The Mach Interface Generator (MIG) [20] was used to implement the RPC mechanism. At initiation, DUST registers the **ARCADEServerPort** with the Environment Manager so that local tasks can use RPC. Then, when a client task wants to request a data unit service, the necessary information is packaged by MIG-provided code as a standard Mach interprocess message, and is sent to the server task by using **msg_rpc()**.

DUST maintains various data structures for task and data unit information. When a request is received on **ARCADEServerPort**, DUST performs the service on behalf of the task. This requires that DUST has access to the client's task port. DUST then returns appropriate information to the client. Since multiple tasks can request services simultaneously, DUST creates multiple threads to handle these requests. The CThreads package [21] was used for this purpose. Currently, however, the threads actually proceed in lock-step. Critical section semaphores will eventually allow all requests to be handled concurrently.

Like OS/2, shared data units are seamlessly updated by mapping them into the address spaces of both the client and DUST. In the Mach implementation, data units are represented as *memory objects*. Currently, the default memory manager is used and the DUST code for Mach is consistent with the OS/2 implementation. In the future, a custom external memory manager could be built which might provide a more efficient implementation.

The default **NetMemoryServer** was used to control the mapping of the memory objects into multiple local tasks' address spaces. When a client task requests a data unit, DUST uses **NetMemoryServer** to create a memory object and to map the object into both his and the client's address space.

The Mach interprocess communication mechanism could have been used to communicate between DUSTs of Mach machines. However, this would have limited Mach's ability to cooperate with other service providers. Therefore, the standard UDP/IP libraries were used to communicate between the server tasks. A separate thread is created by the server task to handle the UDP requests from other data unit service providers.

4. Experiences

Although the three implementations of data units in existing operating systems are not yet fully functional, it has been possible to do some evaluation. Applications which span the various systems have been tested. Most of the data unit features, including cross-machine data unit links and implicit communication and translation have been exercised. These experiments have verified the expected ease of use and efficiency.

The first use of data units on all four systems was an evaluation of a Mandelbrot set. In this multitask application, multiple machines cooperatively calculate values for separate regions of a space. Actual calculation times have been measured for combinations of the four operating systems.

Originally written for ARCADE, the application consists of five task types: user-interface, manager, displayer, progress reporter and worker. The first four handle overhead and worker tasks do the calculation. The manager task sends a data unit to each worker indicating which portion of the space it is to calculate. If these data units cross from a 386-based machine to the System/370, the data is transparently translated. The worker calculates the values for its assigned region, places a data unit link to these values in the original data unit and sends that to the displayer. The manager then assigns a new calculation region. In this implementation, there is no data unit sharing.

For this experiment, one ARCADE machine supported the user-interface, manager, displayer and progress reporter. Workers were then created on various combinations of other machines and the total time to complete the calculation was recorded. The ARCADE and OS/2 machines were all 16 MHz 80386-based IBM PS/2s. Mach was run on a 25 MHz 80386-based PC and VM/CMS executed on an IBM 9370 Model 60.

To establish a baseline, the entire problem was solved on a single machine running ARCADE. The calculation took four minutes and thirty-five seconds. The same problem was then distributed to multiple ARCADE machines and a speed-up curve was calculated. The speed-up was essentially linear up to eight machines at which time a communication problem prevented further testing. Linear speedup is not surprising since the calculation of subregions is independent. A bottleneck seemed to be appearing at the displayer task which was drawing a picture of the region on its screen.

Each of the other operating systems was then evaluated. Since **`run_pgm()`** is not yet fully functional, the worker tasks were started manually. They simply waited for the data unit which told them what to calculate. Table 5 lists the times for

Table 5 - Completion time for a single worker

System	Total Time	Math Time
ARCADE	4:35	4:08
OS/2	4:58	4:08
Mach	7:57	6:12
VM/CMS	8:11	6:58

a single worker task running on each of the operating systems. The first column shows the total time to solve the problem; the second illustrates the variance in compute time caused by different compilers. The slow VM/CMS compute time is due to a variable alignment problem unrelated to data units.

The multiple machine test involved workers on ARCADE, OS/2, Mach and VM/CMS. If the workers progressed at the same rates as in Table 5, the test should have taken one minute and thirty seconds, with the ARCADE task calculating 33% of the space, the OS/2 task 30%, the Mach task 19% and the VM/CMS task 18%. The actual time was one minute and thirty-three seconds, implying a 3% cooperation penalty.

The four implementations of data units allowed cross-system cooperation. Another interesting example is the **rcopy** utility. A user can invoke **rcopy** on any one of the operating systems. This causes a local file to be read into a data unit and the data unit to be sent to an **rcopy** task on another machine. The receiving task will either print the file or save it locally. This convenient bridge is used for printer access from ARCADE as well as for file transfer to the server. It was remarkably easy to write and to port.

5. Conclusions

The central premise of this work is that data units are a valuable cooperation paradigm for multiple operating systems. The expected ease of use and efficiencies have been realized and cross-system applications have been written. The basic services needed for data units were added to three very different operating systems with little difficulty and much common code. However, the one attempt to port the entire ARCADE kernel was not fully successful.

Just as with ARCADE, data units in existing operating systems are a convenient platform for distributed applications. They are easy to write and easy to port. For example, the Mandelbrot calculation was moved from the ARCADE-only world to a set of mixed computers and operating systems with no changes in source code. Data structures were able to transparently span machine boundaries and representation conventions. The **rcopy** utility illustrates how easy it is to move between operating system environments. Data units are not a replacement for the unique distributed services provided by various operating systems. Rather they are a supplement that lets them cooperate as peers.

While the implementation of data units in different operating systems was not easy, it did not pose any insurmountable problems. Although each implementation used different low-level mechanisms, much of the high-level code is the same. For example, all three use the same code for managing data unit locks and updating data unit replicas.

It must be noted that the VM/CMS implementation was designed to support the entire ARCADE service set. The single-threaded nature of CMS made this quite difficult. Although no effort was made to fully port ARCADE to OS/2 or Mach, it is likely that this, too, would have been problematic. However, it has been found that the principle benefits of ARCADE, particularly its support for heterogeneous interconnections, can be realized with just the data unit paradigm.

Work is continuing to refine the implementations of ARCADE and to further test and evaluate the data unit paradigm. As the implementations mature, the inefficiencies in Mach and VM/CMS will be addressed. Also, larger applications, with closer cooperation between elements, will be built. However, it is clear at this point that data units are an extremely promising tool for distributed programming.

6. References

1. Li, K., Shared Virtual Memory on Loosely Coupled Multiprocessors, Ph.D. Dissertation, Yale University, YALEU/DCS/RR-492, September, 1986.

2. Bal, H., Kaashoek, M. and Tanenbaum, A., A Distributed Implementation of the Shared Data-Object Model, *Workshop on Experiences with Distributed and Multiprocessor Systems*, USNIX, Oct., 1989, pp. 1-20.

3. Cheriton, D., The V Kernel: A Software Base for Distributed Systems, *IEEE Software*, April, 1984, pp. 19-42.

4. Chase, J. et al., "The Amber System: Parallel Programming on a Network of Multiprocessors," in *Proceedings of the 12th Symposium on Operating System Principles*, ACM, November, 1989, pp. 147-158.

5. Cohn, D., Delaney, W. and Tracey, K., ARCADE - An Architecture for a Distributed Environment, Department of Electrical and Computer Engineering Technical Report 889, University of Notre Dame, Oct. 1988.

6. Delaney, W. The ARCADE Distributed Environment: Design, Implementation and Analysis, Ph.D. Dissertation, University of Notre Dame, April, 1989.

7. Bertsekas, D. and Tsitsiklis, J., *Parallel and Distributed Computation - Numerical Methods*, Prentice Hall, 1989.

8. Cohn, D. L., Greenawalt, P. M., Casey, M. R. and Stevenson, M. P., Using Kernel-Level Support for Distributed Shared Data, *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, March, 1991, pp. 247-260.

9. Cohn, D., Delaney, W. and Tracey, K., Structured Shared Memory Among Heterogeneous Machines in ARCADE, *Proceedings 1st Symposium Parallel and Distributed Processing*, IEEE, May, 1989, pp. 378-379, also Department of Electrical and Computer Engineering Technical Report 890, University of Notre Dame, Jan. 1989.

10. Ramachandran, U., Ahamad, M. and Khalidi, M., Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer, *Proceedings 1989 International Conference on Parallel Processing*, Volume II, August, 1989, pp. 160-169.

11. Ramachandran, U. and Khalidi, M., An Implementation of Distributed Shared Memory, *Workshop on Experiences with Distributed and Multiprocessor Systems*, USENIX, Oct., 1989, pp. 21-38.

12. Kessler, R. and Livny, M., An Analysis of Distributed Shared Memory Algorithms, *Proceedings 9th International Conference on Distributed Computing Systems*, IEEE, June, 1989, pp. 498-505.

13. Bisiani, R. and Forin, A, Multilanguage Parallel Programming of Heterogeneous Machines, *IEEE Transactions on Computers*, Vol. 37, No. 8, Aug. 1988, pp. 930-945.

14. Cheriton, D., Problem-Oriented Shared Memory: A Decentralized Approach to Distributed System Design, *Proceedings 6th International Conference on Distributed Computing Systems*, IEEE, May, 1986, pp. 190-197.

15. *VM/ESA Introduction Release 1*, IBM Corporation, June, 1990.

16. Duncan, R., *Advanced OS/2 Programming*, Microsoft Press, 1989.

17. Tevanian, A. and Rashid, R., Mach: A Basis for Future Unix Development, Technical Report CMU-CS-87-139 Carnegie-Mellon University, June, 1987.

18. Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M., Mach: A New Kernel Foundation for UNIX Development, *Proceedings of Summer Usenix*, July, 1986.

19. Smith, E. R., "A New Distributed File System", Master's Thesis, University of Notre Dame, August, 1990.

20. Draves, R. P., Jones, M. B., and Thompson, M. R., "MIG - The MACH interface Generator", Department of Computer Science, Carnegie-Mellon University, November, 1989.

21. Cooper, E. C., Draves, R. P., "C Threads", Department of Computer Science, Carnegie-Mellon University, September, 1990.

Experiences in Parallel Performance Measurement: The Speedup Bias

Frederick Wieland
Peter Reiher

Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 91109
fred@ruby.jpl.nasa.gov

David Jefferson

University of California
Los Angeles, CA
jefferson@lanai.cs.ucla.edu

Abstract. *Experience in measuring the performance of parallel simulations developed at the Jet Propulsion Laboratory has shown that speedup is systematically overstated by a phenomenon we call speedup bias. Unlike other speedup overstatements which are caused by distributing overhead or by hardware effects, speedup bias is caused by the fact that a parallel program executed sequentially runs less efficiently than an equivalent program implemented for a sequential processor. In this paper, we use the Concurrent Theater-Level Simulation (CTLS) parallel simulation to present the first empirical evidence for the existence of speedup bias. We also introduce two new terms, implementation speedup and model speedup, which describes two ways of measuring performance. Although implementation speedup is the traditional measurement, we argue that model speedup is more accurate; speedup bias is simply the ratio of implementation speedup to model speedup. We find that, for the CTLS model, the speedup bias is 3.1; that is, the traditional metric of speedup overstates the true parallel performance gains by a factor of 3.1. This result is important because it confirms the existence of speedup bias, and provides the first empirical test of its magnitude.*

1. Introduction

This paper presents the first empirical measurement of a commonly-accepted phenomenon which we call *speedup bias*. Speedup bias is the overstatement of speedup which occurs when the sequential time against which speedup is measured is the sequential execution time of a *parallel* program. Speedup bias exists because a given program cannot be simultaneously optimized for both parallel and sequential performance. Because a given parallel program is designed for good *parallel* performance, comparing its sequential run time to its parallel run time will overstate its speedup by some unknown amount.

We will introduce two new terms associated with speedup bias. Both terms describe the type of speedup which is measured. The first term, called *implementation speedup*, describes the speedup measured when both the sequential and parallel execution times use a parallel implementation. Implementation speedup is the most common form of speedup published in the parallel literature. The second term, called *model speedup*, describes a situation in which a given model has been implemented twice, once for efficient parallel performance and once for efficient sequential performance. The speedup ratio in this case accurately reflects the performance gains when parallelism is applied to the model itself. These ideas are further discussed and refined in section 4.

The ideas and measurements in this paper are the result of our experience with the Concurrent Theater-Level Simulation (CTLS). CTLS is a large (60 K lines of code) parallel combat simulation developed at the Jet Propulsion Laboratory for the U. S. Army. Although the empirical measurements in this paper are applicable only to CTLS, speedup bias is a general phenomenon applicable to any parallel computation. It is particularly acute in parallel discrete-event simulations such as CTLS, because the structure of such simulations is quite different from an equivalent sequential simulation.

2. Existence of Speedup Bias

Recent literature on the performance of parallel discrete-event simulations have hinted at the existence of speedup bias. Fujimoto [Fuji 90] states that the logical process method of programming, which requires parallel programmers to partition shared memory into disjoint sets, leads to programs in which data structures are both distributed and replicated. This approach yields programs which are significantly more complicated than their sequential counterparts. Fujimoto implies that, although replicated and distributed data structures improve parallel performance, they lead to longer run-times for a sequential program.

Studies at JPL [Wiel 89A] have similarly shown that parallel simulations must be carefully designed if they are to extract any speedup at all. In particular, the style of message passing between objects which share information must be carefully chosen so as to minimize message traffic and maximize parallelism. Again, any design choices made to improve parallel performance will likely result in slower sequential executions, which bias the resulting speedup measurements.

In [Heln 89], a general study of parallel performance issues applicable to any parallel program performance measurement is presented. The study cites several examples of ancillary issues that can artificially increase the speedup. For example, the presence of more caches as the number of processors increases will cause memory references to complete disproportionately faster. Also, the communication bandwidth and total main memory available increases with the number of processors. Additionally, overhead of the underlying operating system (such as queueing insertions and deletions) is distributed as the number of processors increases. All of these (and other) effects can artificially inflate the speedup numbers. In general, if the total amount of work done by the sequential system is greater than the total amount of work done by the parallel system, then the speedup numbers are inflated, often to the point of superlinearity.

The speedup bias presented herein is different than the inflated speedup effects discussed by Helnhold. Speedup bias assumes that the underlying operating system has been implemented efficiently, both for sequential processing and parallel processing (a point which we discuss in section 3). The speedup is artificially inflated not because the parallel processor has additional resources, but because the sequential run-time has the added overhead of distributed and replicated data structures. In effect, the speedup is biased because the total amount of work done by the sequential system is greater than the amount of work that would be done if the system had been implemented expressly for sequential processing. Thus the numerator of the speedup equation is inflated by the extra cost of distributing and replicating data structures, which would otherwise be absent, resulting in the speedup bias.

3. Time Warp Operating System

For the remainder of the article, we will focus on the performance of parallel simulations. Parallel simulations are a class of parallel computations in which a simulation of a physical system is executed in parallel. Note that parallel simulation is *not* the simulation of a parallel system, but rather the parallel execution of some simulation. Simulations of any type, whether parallel or sequential, require some type of underlying simulation engine. For this study, we used two different simulation engines. The first simulation engine, called the Time Warp operating System (TWOS), was developed at JPL solely for executing simulations in parallel. TWOS is a complete implementation of the theory of Virtual Time [Jeff 85], including all synchronization mechanisms, such as rollback, message cancellation, and memory management/flow control, as well as a GVT algorithm which scales as log N (N being the number of processors) [Bell90]. Time Warp has been extensively studied in the literature; see, for example, [Rei90A Rei90B, Pres90, Fuji90, Wiel89 B]. Although TWOS currently includes a dynamic load management facility [Rei90C], when the measurements presented herein were taken (January, 1991), load management was disabled.

The second simulation engine, called the Sequential Simulator (SS), was developed at JPL solely for executing simulations sequentially. The SS provides the same interface to a simulation that is provided by TWOS, although the internals are different. The SS does not incur any of the concurrency control overhead required by Time Warp (rollback, antimessages, etc.), and its event list is implemented as a splay tree, which is the best general purpose event list known. As a result, the SS executes the simulation as fast as possible, while being compatible with TWOS.

Because TWOS is built specifically for parallel processing, and the SS is built specifically for sequential processing, we claim that there is no speedup bias in the simulation engines themselves. All of the measured speedup bias is attributed to the simulation implementation itself, which was built solely for parallel execution.

4. Definition of Terms

We are now in a position to define speedup bias more rigorously. For sufficiently large simulations, such as the kind that would most benefit from parallel processing, two groups of people are usually involved: the *users* and the *developers*. The users are those who run and use the output of the simulation, whereas the developers are those involved in its design and implementation. For small simulations, the users are the developers. A simulation *model* is an informal combination of prose and mathematics that describes the system to be simulated. The simulation *observables* are data output from the model which are of interest to the users. It is the user community's responsibility to define the model along with its associated observables. Usually this step is done with some input from the developers to determine the feasibility and costs of the various requirements.

An *implementation* of a model is its software realization. It is embodied as a set of programs and data files that, when executed, yield the required observables. In designing the implementation, a developer might produce many intermediate stages of software design, including data flow diagrams, structure charts, structured text, etc. [Your82]. Although these intermediate stages may be useful to the user community, we will consider them part of the software implementation and not part of the simulation model.

A *sequential implementation* is one wherein the developer has explicitly targeted the system for a sequential processor. Likewise a *parallel implementation* is one expressly designed to work well on a multiple-processor system.

We are now in a position to define *implementation speedup* and *model speedup*. Implementation speedup is the ratio of sequential wall-clock time to parallel wall clock time when there is only one parallel implementation of the model. Model speedup is the ratio of sequential to parallel time when an efficient sequential implementation and an efficient parallel implementation of the model are used. *Speedup bias* is simply the ratio of implementation speedup to model speedup, and measures the factor by which implementation speedup overstates model speedup.

We shall argue in section 9 that model speedup is the only speedup metric which accurately reflects the true performance gains of parallelism, although it is the implementation speedup that is commonly presented in the literature. Because developers do not have the time and resources to simultaneously produce both a sequential and a parallel implementation of a given model, when measuring speedup of a parallel implementation, it is the implementation speedup that is being measured. This speedup does not truly reflect the gains of parallel processing, because a parallel implementation does not run as efficiently on one processor as would a sequential implementation of the same model.

5. The Parallel Environment

The particular parallel program on which we measured speedup bias is a parallel simulation known as *CTLS*. CTLS ("Concurrent Theater-Level Simulation") is a theater-level ground combat simulation developed for the U. S. Army, and uses TWOS as its vehicle for parallel execution. Both CTLS and TWOS are are coded in the "C" programming language and use the BBN Butterfly GP1000 parallel processor. The GP1000 contains 84 processors, each consisting of a 68020 CPU and 4 megabytes of memory. Although the speedup bias measurement uses this particular hardware and software platform, the phenomenon of speedup bias is applicable to *any* parallel computation in which speedup is being measured. However, for different hardware and software platforms, we would expect the speedup bias to be different than the one measured here.

The CTLS model is complex. Briefly, it consists of several different subsystems: a high-resolution ground subsystem, which simulates a military command hierarchy from Company to Theater level; a simple air subsystem, which simulates air reconnaissance flights for intelligence gathering, and incorporates the effects of ground-air attrition; a combat subsystem, incorporating both indirect and direct fire via the Lanchester linear and Lanchester squared attrition process; and a logistics subsystem, which allocates scarce resources and delivers them via explicitly modelled convoys. The version of CTLS used in this study contains all of these subsystems, while current CTLS versions under development include additional subsystems that add fidelity and resolution to the model.

The parallel implementation of this model consists of 160 instances of five types of objects, and a variable number of instances of a sixth object type. The *planner* object is responsible for interpreting a scripting language, which provides a flexible run-time method for analysts to control the simulated scenario. The planner is also responsible for part of the logistics subsystem. The *combat* object is responsible for movement, engagement, resupply, and periodic reporting of its situation to its superior. The *mover* object is responsible for moving supplies from one location to another, as part of the logistics subsystem. The *air planner* and *flight group* objects are responsible for the air subsystem, including carrying out a user-defined missions, reconnoitering, and then flying back and relaying back the information. The flight group is also responsible for alerting ground anti-aircraft combat objects of its existence, and for computing ground-air attrition.

6. CTLS Proximity Detection

The sixth object type, which can have a varying number of instances, is the *sector* object type. The sector object performs an algorithm known as *proximity detection.* Proximity detection involves determining when one unit can sense another unit, or, equivalently, determining when two units are close enough for some type of interaction. Any board-based simulation requires some form of proximity detection. Many different styles of proximity detection have been used, and each style is tailored to a specific model or a specific set of requirements [Wiel90, Conk90, Lub90, Hon89].

Although the proximity detection subsystem is only one of six different subsystems in the CTLS model, it is the dominant subsystem in terms of performance. Studies of the critical path of CTLS have shown that over eighty percent of the events on the critical path of the simulation are proximity detection events [Wiel91].

Instances of the sector object type are known as "sectors," because they serve to partition the gameboard. CTLS allows the number of sectors to vary without any effect on the model observables. The observables are identical whether CTLS is run with one sector covering the whole gameboard, or one thousand sectors tiling the gameboard. Neither the value of the observables, nor their precision, nor their output frequency changes as the number of sectors is changed.

What does vary with the number of sectors is the performance of the implementation. Each sector executes in parallel with the other sectors, and performs the following two computations:

> (1) They determine when a unit within their borders crosses into a neighboring sector, which requires communication between sectors.
> (2) They determine when a unit within their borders interacts with another unit within their borders, which requires computation within the sector. The computation involves solving a quadratic equation for each pair of units within the sector.

Both algorithms (1) and (2) are affected by the number of sectors on the gameboard. If the number of sectors is increased, then units cross sector boundaries more often, but there are, on average, fewer units in each sector. Therefore, as the number of sectors is increased, there is more communication between sectors and less computation within them.

7. Performance Experiments

Granularity, or the ratio of the mean computation time per event to the mean overhead per event, is a major performance parameter for parallel simulations [Wiel87, Sokol 92]. Because by changing the number of sectors we are, in effect, changing the granularity of the computation, we are in a unique position to tune the number of sectors so that the resulting system is optimized for either sequential or parallel execution.

All the measurements shown herein were taken on a BBN Butterfly GP1000 on either one processor (for the sequential data) or seventy processors (for the parallel data). The GP1000 consists of Motorola 68020 processors, each with 4 Mbytes of memory. The Mach operating system was running beneath TWOS. Although Mach provides virtual memory, no page faults occurred on any of the parallel executions. Furthermore, on the sequential runs, a maximum of 278 page faults (about 10 seconds of execution time) was observed over a forty-minute run. Thus we conclude that the page faults were not biasing the results in any serious manner.

Figure 1 presents the execution time for CTLS run both sequentially and also in parallel on seventy processors. The number of sectors varies from 1 to 1,024, and are plotted logarithmically along the x-axis. The shape of the parallel curve is influenced by the tradeoffs between increased computation within a sector with its resulting higher granularity but lower parallelism vs. increased communication among sectors and its associated higher parallelism but lower granularity. The shape of the sequential curve is influenced by the lower computational complexity within a sector vs. higher queueing and event scheduling overhead. These tradeoffs fight each other in different ways, which results in the fastest run time at eight sectors for the sequential runs, vs. 800 sectors for the parallel runs.

The difference between the two curves illustrates the impossibility of producing an implementation which runs fast both in parallel and sequentially. A developer targeting the implementation for a sequential processor would have no need for 1,024 sectors; indeed, eight would suffice. Conversely, a developer targeting the implementation for a parallel processor would want to maximize parallelism with as many sectors as possible; in this case, 800 sectors is best. Unless the system were implemented in a manner to let the number of sectors vary, a developer would have to fix the number of sectors at some reasonable value. For a sequential implementation, that "reasonable value" is different from that for a parallel implementation.

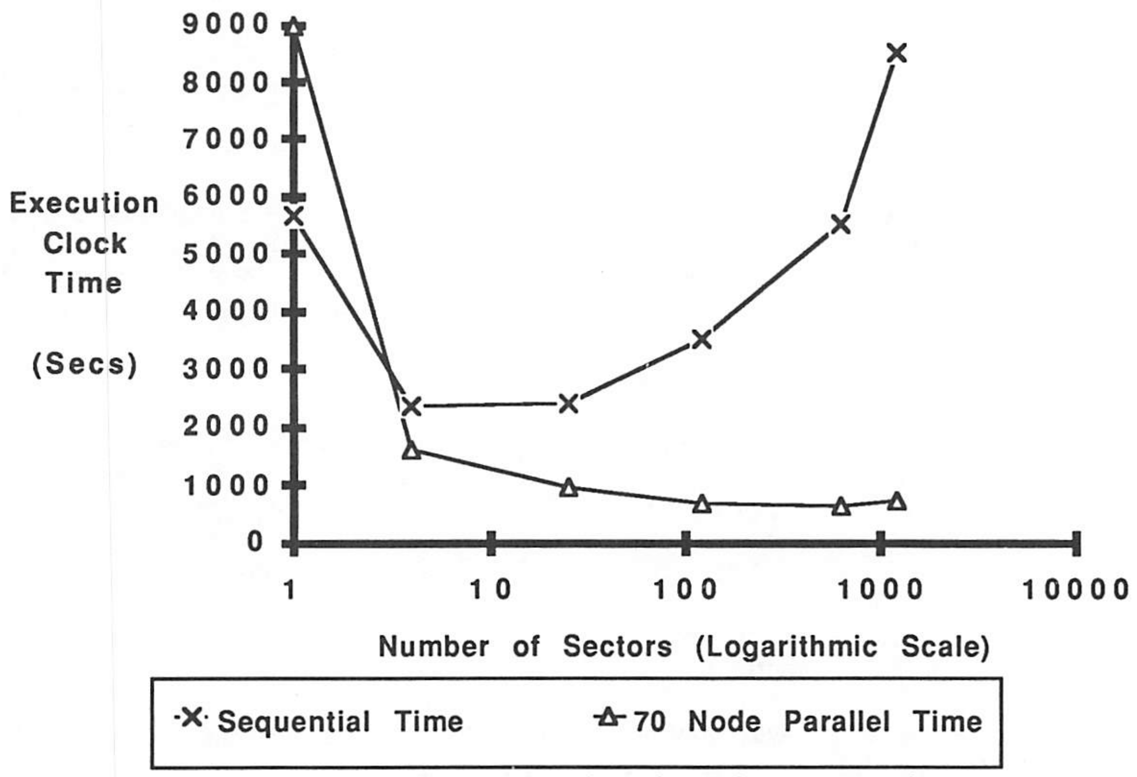


Figure 1. Execution time as a function of number of sectors.

8. Model Speedup and Implementation Speedup

In measuring implementation speedup, we would fix the number of sectors at some value and then measure the ratio of the sequential execution time to the parallel execution time using that fixed number of sectors. Model speedup, however, requires that an efficient *sequential* implementation of the model be the yardstick against which speedup is measured. Figure 2 plots both the implementation speedup and the model speedup for the data presented in figure 1. The implementation speedup is just the ratio of the sequential to the parallel run time as the number of sectors is varied. The model speedup, however, fixes the numerator at the run-time for an eight sector sequential system, because at eight sectors the sequential time is the fastest. The denominator of the model speedup is just the parallel run time, which varies with the number of sectors.

Mathematically, if $t_S(N_S)$ is the sequential run time for N_S sectors, while $t_p(N_S)$ is the parallel run time for N_S sectors, then:

Implementation speedup = $t_S(N_S) / t_p(N_S)$

Model speedup = $t_S(8) / t_p(N_S) = 2376 / t_p(N_S)$

Note that the implementation speedup is the normally measured speedup: the same program is executed both sequentially and in parallel, and the ratio of the run times is computed. However, the model speedup fixes the numerator at 4 sectors (with a run time of 2376 seconds), and allows the parallel system to vary with the number of sectors. The model speedup is fixing the sequential time at the number of sectors which produces the fastest run time. The model speedup is measuring the performance gains due to parallelizing the model; the implementation speedup is measuring performance gains as a particular, fixed implementation is run on more processors. Because, as explained above, the observables in CTLS do not change as the number of sectors are varied, we are comparing equivalent models even though the sequential and parallel systems might utilize a different number of sectors.

Figure 2 plots both the implementation speedup and the model speedup as the number of sectors are varied.

9. Implications for Parallel Performance

The implication of these measurements on interpreting parallel performance results is dramatic. The highest implementation speedup was 12 using 1,024 sectors, whereas the highest model speedup was 3.8 using 800 sectors. The speedup bias for this model is thus 12 / 3.8 = 3.1, which means that the implementation speedup measurement can overstate the model speedup by a factor of 3.1. A paper describing the performance of CTLS would have been correct in reporting the implementation speedup of twelve on seventy processors, in the sense that generally accepted methods for measuring speedup would have been used in reporting that number. Although such a speedup would not be considered especially high, it overstates, by a factor of 3.1, the actual performance gain a parallel implementation of CTLS would have had over a sequential implementation of the model.

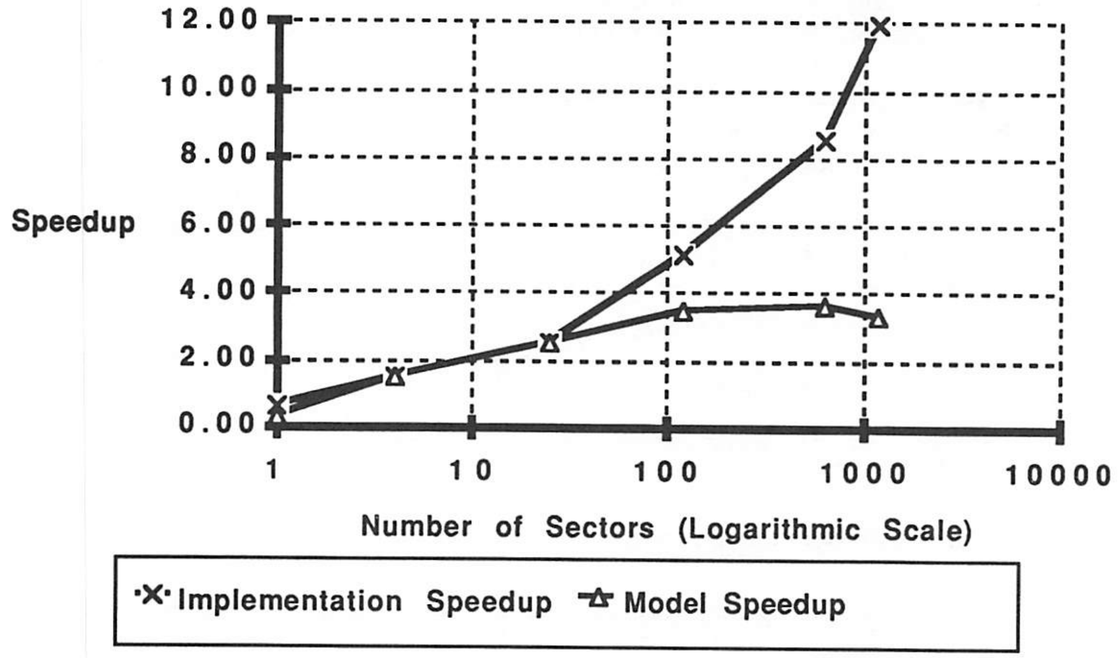


Figure 2. Speedup vs. Sectors

This data also sheds light on the commonly accepted research practice of structuring a system to maximize speedup. In this specific example, the minimum sequential execution time occurs with eight sectors, whereas the minimum parallel execution time occurs with 800 sectors. Curiously, the maximum implementation speedup occurs with 1,024 sectors. Thus if the goal of CTLS were to maximize speedup, then one would measure the implementation speedup using a fixed 1,024 sector system. However, such a measurement is misleading for two reasons. First, it overstates (by a factor of 3.1) the true performance gains of parallelism. Secondly, it uses run times for both the sequential and parallel systems which are not the fastest run times in either case. At 1,024 sectors (and beyond), both the sequential and parallel execution times are degrading. The sequential time is degrading at a faster rate than the parallel time, leading to the absurd conclusion that speedup is unbounded as the number of sectors increases.

Besides the highest implementation speedup of 12 at 1,024 sectors, the data reveal two other important speedup measurements. One of them is the implementation speedup where the parallel execution time is the fastest. Because the fastest parallel execution time occurs at 800 sectors, the implementation speedup at that point is 8.6. Although this speedup number is slightly more accurate than just measuing the highest speedup available, it also overstates the performance gains of parallelism because the sequential time used in its computation is not the fastest sequential time.

The third and best speedup number is the model speedup where the denominator $t_p(N_s)$ is at 800 sectors, which is the fastest parallel execution time. This measurement is comparing identical models, although the numerator uses the fastest sequential time while the denominator uses the fastest parallel time. For this measurement, the number of sectors is different in the sequential and parallel case, although the resulting speedup is valid because the model observables are identical in either case. The resulting speedup number of 3.8 yields the insight that CTLS can run a factor of 3.8 times faster on seventy processors than

Measurement	Number of Sectors	Type of speedup	Value
Maximum speedup	1,024	Implementation	12
Speedup where parallel run time is fastest	800	Implementation	8.6
Fastest seq. time/ fastest par. time	800 (parallel) 8 (sequential)	Model	3.8

Figure 3. Summary of the three different speedup measurements

it can with a properly structured sequential implementation on one processor. While the model speedup will always be lower than any of the implementation speedups, and therefore appears less attractive, it nevertheless accurately reflects the relative performance choices that users and developers have in regard to simulation implementation. The choices are either to implement an efficient sequential simulation, or to implement an efficient parallel simulation. Only the model speedup number yields an insight into the relative attractiveness of these two choices.

In this study, we varied only the proximity detection subsystem in CTLS. Although the proximity detection subsystem is the most dominant of the six subsystems in CTLS, the other five subsystems are also implemented in a parallel manner. Had these other subsystems been recoded for maximum sequential performance, then the model speedup would have been *less* than the 3.8 reported here, and the speedup bias would have been greater than 3.1.

10. Practical Considerations

Although there are a variety of different possible speedup measurements, we have argued that only one of them (the model speedup) is useful for decision making. The method by which a researcher would obtain model speedup in the absence of two separate implementations is currently unknown. In this study, the parallel implementation was flexible enough to vary a key performance parameter without affecting the model observables. In the absence of such flexibility, the researcher would have to make an intelligent estimate as to the amount of additional overhead added to the sequential run by the presence of parallel processing overhead. Because this whole subject of speedup bias is relatively new, there are no historical databases or experiences other than the one outlined here to help the researcher with his estimate.

11. Conclusions

Measuring speedup by computing the speedup ratio of two fixed implementations is called "implementation speedup," and yields misleading information. One type of implementation speedup is that which yields the highest speedup measured; for CTLS this occurs at 1,024 sectors and yields a speedup of 12 on 70 processors. Another type of implementation speedup is that implementation which uses the fastest parallel execution time. For CTLS, this occurs at 800 sectors and yields a speedup of 8.6. Finally, the speedup measured when the fastest sequential implementation is compared to the fastest parallel implementation is called the "model speedup." For CTLS, the fastest sequential implementation occurs at eight sectors, and the fastest parallel implementation occurs at eight hundred sectors. The CTLS model speedup is 3.8. Thus the speedup bias, or the overstatement arising from using the highest implementation speedup rather than the model speedup in reporting results, is a factor of 3.1.

The main conclusion of this study is that, of the three speedup metrics reported here, only the model speedup makes any sense for users and developers of simulations. Faced with a choice of implementing either a sequential simulation or a parallel simulation, developers need to estimate the model speedup in order to make a meaningful choice. Relying on any type of implementation speedup results in making critical decisions with misleading data.

Acknowledgements

We would like to thank JPL personnel Paul Springer, Gerry Fleischer, John Gieselman, Larry Hawley, and John Wedel, as well as managers Jack Tupman and Herb Younger. We wish to thank Phil Louer and John Shepherd of the U. S. Army Concepts Analysis Agency for their ongoing interest and support of this project.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Army Model Improvement and Study Management Agency through an agreement with the National Aeronautics and Space Administration. Reference herein to any specific commercial product does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, Calfornia Institute of Technology.

[Bell90] Steven Bellenot, "Global Virtual Time Algorithms," *Proceedings of the 1990 Distributed Simulation Conference,* Society of Computer Simulation, 22(3), San Diego, January, 1990

[Conk90] Darrell Conklin, John Cleary, and Brian Unger, "The Sharks World," *Proceedings of the SCS Conference on Distributed Simulation,* San Diego, CA. Society for Computer Simulation, 22(1), January, 1990

[Fuji90] Richard Fujimoto, "Parallel Discrete-Event Simulation," *Communications of the ACM,* 33(10), October, 1990.

[Heln 89] Helnbold, D. P. and McDowell, C. E., "Modeling Speedup(N) > N", *Proceedings of the 1989 International Conference on Parallel Processing,* vol III, August 1989, pp 219-225.

[Hon89] Philip Hontalas, *et. al.,* "Performance of Colliding Pucks Simulation on the Time Warp Operating System," *Proceedings of the SCS Conference on Distributed Simulation,* Tampa, Florida. Society for Computer Simulation 21(2), March, 1989.

[Jeff 85] David Jefferson, *et al.,* "Virtual Time," ACM *Transactions on Programming Languages and Systems,* Vol 7 No. 3, July 1985.

[Lub90] Boris D. Lubachevsky, "Simulating Colliding Rigid Disks in Parallel Using Bounded Lag Without Time Warp," *Proceedings of the SCS Conference on Distributed Simulation,* San Diego, CA. Society for Computer Simulation, 22(1), January, 1990

[Rei90A] Peter Reiher, "Parallel Simulation Using the Time Warp Operating System," *Proceedings of the 1990 Winter Simulation Conference,* December, 1990.

[Rei90B] Peter Reiher, Frederick Wieland, Philip Hontalas, "Providing Determinism in the Time Warp Operating System — Costs, Benefits, and Implications," *Proceedings of the IEEE Workshop on Experimental Distributed Systems,* October, 1990.

[Rei 90C] Peter Reiher, David Jefferson, "Virtual Time Based Dynamic Load Management in the Time Warp Operating System," *Transactions of the Society for Computer Simulation,* 7(2), June, 1990.

[Sokol 92] Lisa M. Sokol, Paula A. Mutchler, and Jon B. Weissman, "The Role of Event Granularity in Parallel Simulation Design," *Proceedings of the SCS Conference on Distributed Simulation,* Newport Beach, CA, Society for Computer Simulation 24(3), January, 1992.

[Wiel90] Frederick Wieland, Lawrence Hawley, Leo Blume, "An Empirical Study of Data Partitioning and Replication in Parallel Simulation," *Proceedings of the 1990 Distributed Memory Computing Conference,* March, 1990.

[Wiel89 A] Frederick Wieland and David Jefferson, "Case Studies in Serial and Parallel Simulation," *Proceedings of the 1989 International Conference on Parallel Processing,* vol. 3, August, 1989, pp. 255-258.

[Wiel89 B] Frederick Wieland, *et. al.,* "The Performance of a Distributed Combat Simulation with the Time Warp Operating System," *Concurrency: Practice and Experience,* 1(1), September, 1989.

[Wiel87] Frederick Wieland, Lawrence Hawley, and Abe Feinberg, "Implementation of a Distributed Combat Simulation using Time Warp," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications,* ACM, January, 1988.

[Your82] Edward Yourdon, *Managing the System Life Cycle,* Yourdon Press: 1982.

Supporting User-Level Exception Handling on a Multiprocessor Micro-Kernel: Experiences with PLATINUM

Robert J. Fowler and Leonidas I. Kontothanassis *
Department of Computer Science
University of Rochester
Rochester, New York 14627
fowler@cs.rochester.edu, kthanasi@cs.rochester.edu

Abstract

We describe the facilities provided by PLATINUM, a multiprocessor micro-kernel, to support user-level exception handling. The principal design goal for these facilities is to provide a very simple mechanism with sufficient flexibility and efficiency to experiment with the implementation of a wide variety of exception-driven, user-level services in a parallel system. These services provide conventional exception handling, an interface between parallel programs and debugging or performance monitoring tools, user-level paging and I/O services, and support for user-level lightweight process packages. The exception handling facility is based on fast message passing using a reconfigurable hierarchy of mailboxes. The mechanisms we describe have performance comparable to their uniprocessor signal handling under Unix, and retain that performance when used on a multiprocessor to support the kind of exception-driven services demanded by a parallel programming environment.

1 Flexible Exception Handling on Multiprocessors

The practice of structuring an operating system as a relatively small kernel (popularly called a "micro-kernel") augmented by a collection of server processes continues to be an active topic of research [Ras86, Mul87]. Although the modularity of this approach yields software engineering advantages such as improved comprehensibility, ease of maintenance, and reliability, the increased number of inter-module control transfers in such a system can put it at a performance disadvantage with respect to more monolithic designs[ALBL91]. On a multiprocessor, however, the modularity of the micro-kernel approach can be exploited to take advantage of parallelism by replicating and distributing servers across multiple processors. Because many operating system services are exception-driven, we address the issue of providing a kernel-defined facility to support modular and efficient exception handling in a multiprocessor micro-kernel system. In this paper we present a set of requirements for such facilities, a description of our solution to the problem, an evaluation of that solution, and a discussion of remaining problems and directions for future research.

The combination of the micro-kernel design and the emphasis on logical and physical parallelism requires exception handling that is flexible, efficient, and allows maximum parallelism on a multiprocessor. Some of the many potential applications for user-level exception handling on a multiprocessor include:

1. Providing an error recovery and/or cleanup mechanism,
2. Implementing "software interrupts" used in I/O and communications protocols at user level,

*This material is based upon work supported by the National Science Foundation under Grant number CDA-8822724. The Government has certain rights in this material.

3. Supporting user-level scheduling of lightweight process models [ABLL91, MSLM91],
4. Implementing virtual memory paging at user level [RTY+88],
5. Using memory management operations for non-traditional purposes [AL91], and
6. Constructing interfaces between debugging and performance monitoring tools and their target programs.

In some systems, handling an exception at user level is perceived to be a rare event that is not critical to system performance. In contrast, we anticipate that such events will be extremely common in parallel systems with user-level virtual memory operations, sophisticated user-level thread packages, or debugging tools that allow the user to set conditional breakpoints in multiple processes on multiple processors. User-level exception handling must therefore be efficient enough that it can be employed profitably in these applications.

There are many potential applications for efficient exception handling and there are many alternative strategies for implementing exception handling mechanisms. Rather than choosing a particular fixed strategy at the time the kernel is designed, we contend that it is appropriate to leave the choice to the programming environment or to the user. As long as the performance-critical cases are not adversely affected, it is appropriate to defer some decisions to run time and, in some cases, to change them dynamically.

While the exception handling mechanism must be flexible, a programmer should not have to confront it in its full generality until the flexibility is really needed. We therefore require that the system provide a default exception handling policy that will serve in most situations, but which can be replaced easily and incrementally as the need arises.

Our experiments in the design and implementation of kernel-level mechanisms to support handling exceptions at user-level so as to meet all of our perceived requirements were conducted with PLATINUM. PLATINUM (an acronym for "PLATform for Investigating Non-Uniform Memory") is a multiprocessor kernel designed to support research on locality management issues on Non-Uniform Memory Access Cost (NUMA) multiprocessors such as the BBN Butterfly GP1000. Our intention was to expend a relatively small amount of effort to create an easy-to-use and easy-to-modify execution environment in support of our planned experimental research. We have therefore tried to keep the system and its interfaces small. The kernel provides low-level address space and thread management mechanisms, communication via mailboxes, and basic device drivers. While PLATINUM is based on the Mach virtual memory model, the rest of the kernel interface can be considerably "leaner" than Mach since it is designed to run on a single-user multi-processor system, whereas Mach supports multiple systems connected across a network. In keeping with the general design philosophy, we sought a single exception handling mechanism that was very efficient, general, and simple to implement.

A Framework for Exceptions and Exception Handling

Behaviorally, an exception is an interruption of the normal order of instruction execution on a processor. An exception can occur when an attempt to execute some an instruction causes a condition (*e.g.* a page fault) that prevents the program from continuing correctly. Events generated by devices such as timers, I/O devices, or other processors in a parallel system can also cause exceptions. Generally, exceptions of the former kind are known as *traps* while those of the latter kind are known as *interrupts*.

We divide the life of an exception into four phases.

- **Detection:** The system recognizes that an exceptional event has occurred.
- **Reporting:** The system decides what kind of exception has occurred, constructs a data structure that describes the event, and dispatches it to an appropriate handler.
- **Handling:** The handler either terminates the target thread of control or takes the necessary actions to put the system in a state from which the thread can continue. In addition to

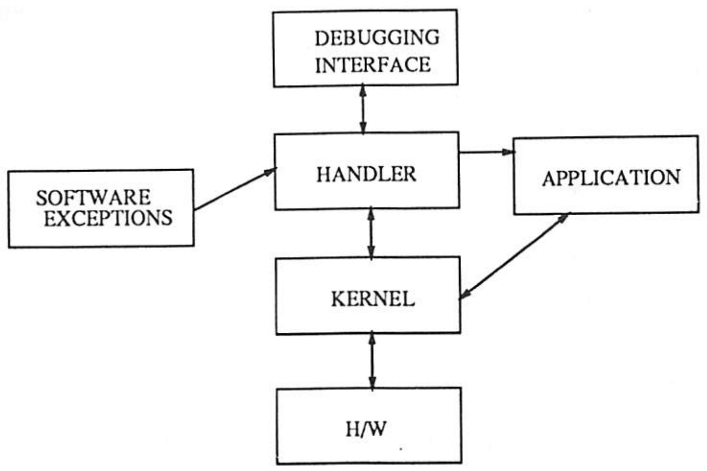


Figure 1: A proposed general model for exception handling

the information recorded by the event detection and reporting mechanism, the handler also requires a means of examining and manipulating process state [Els88] as well as other resources in the system.

- **Continuation:** If the target thread can be resumed, it does so from the state established by the handler.

The handlers of various types of exceptions require differing degrees of privilege and of isolation from the application program. Interrupts and traps that require the handler to access physical resources such as device control registers or memory management hardware need to have at least part of their handlers in the operating system kernel. Some traps (*e.g.* segmentation faults) need to be handled from a safe vantage point outside the writable address space of the application. Some are recoverable (*i.e.* overflows, page faults, or timer interrupts) and the system should allow the application to install handlers that will perform the necessary correction steps. While in principle all exceptions could be handled from the safe vantage point of a separate protection domain, this strategy can be expensive because of the cost of context switches and of inter-domain operations for examining and manipulating the state of a target program. This added cost can be prohibitive if exceptions are not "rare events". This performance consideration will become more important as more applications for user-level exception handling are found.

In a layered system, there may be many potential levels of exception handling that will perform variations on each of the above phases. For example, in VAX VMS [LE80] the handling of a single exception may be distributed across many levels including a low-level handler for the "hardware interrupt", a "software interrupt" handler in the operating system kernel, and another handler in user mode. Furthermore, the user program can activate and deactivate additional user-level handlers dynamically.

Figure 1 illustrates the structure of a default exception handling mechanism in PLATINUM. The hardware detects the exceptions. The kernel completes the detection phase. Some exceptions, such as device interrupts and some page faults, are handled directly by the kernel. The rest are reported to a handler at user level. The user-level handler may take standard actions to correct or terminate the faulting thread of control. It may also pass the exception to code installed in the application program. In some cases it may forward the exception to another thread of control. If debugging is enabled, for example, the handler may cooperate with a debugger host server program that communicates over a network with an interactive debugging tool running on a remote workstation. Software interrupts, such as those from file and communication services, may enter the system at several points in this hierarchy.

The number of "reasonable" alternatives for exception reporting can be very large on a multiprocessor with multi-threaded and multiple address space applications. Some exceptions can be handled on a per-thread basis, some should be handled on a per-address space basis, and some should be handled on a per-application basis [BGH+88]. In object-oriented programs, a handler might be allocated to serve all instances of a particular class, a subset such as the instances on a particular processor, or even a single instance. It is possible to execute a handler on any of the available processors, and some appropriate choice needs to be made. Finally, the existence of parallelism adds the potential of having multiple exception handlers executing concurrently. Given this wide variety of alternatives, it is important that it be possible for an application program to choose dynamically among them. We provide a mechanism that allows user-level code to create hierarchies of handlers and to be responsible for dispatching exceptions among them.

2 Related Work.

2.1 Applications of User-Level Exception Handling.

In addition to giving user programs a flexible mechanism for handling their own errors, we focus on three other applications of user-level exception handling on multiprocessors. All three of these applications require that the exception-handling mechanism be efficient and flexible when implemented on a multiprocessor.

Parallelism in exception handling can be used to great advantage for the debugging of multi-threaded programs. For example, a *conditional breakpoint* in a thread of control can be implemented as an ordinary breakpoint in which a debugger agent passes control to the user interface part of the debugger only if the specified condition is satisfied [FLMC88]. Otherwise, the thread is resumed without any further communication. If all of the threads of a parallel program have conditional breakpoints activated, it is preferable to have one debugger agent per physical processor rather than to serialize the condition testing in a single agent that serves all of the threads.

A recent trend in multiprocessor systems is two-level scheduling, in which an operating system kernel schedules threads of control that in turn act as virtual processors scheduling user-level threads. The virtual processor handles a variety of software interrupts used to signal actions and events in the underlying kernel, including the completion of I/O operations, timer interrupts, and scheduling actions affecting the virtual processors. Although different kernel interfaces for supporting two-level scheduling have been proposed [ABLL91, MSLM91], they all depend on an efficient software exception, or upcall, mechanism. To support experimentation with a wide variety of these mechanisms, we require a general and efficient exception handling mechanism that gives users the ability to define new exception types and to install handlers for them.

There are several uses for the user-level handling of exceptions associated with memory management. The Mach memory management system [RTY+88] supports the installation of *external pagers* that operate outside the kernel. The kernel forwards page fault exceptions to these user-level programs that in turn can be used to implement multiple custom paging mechanisms and policies. Recently Appel and Li [AL91] have surveyed several applications of user-level virtual memory operations including garbage collection, checkpointing, distributed virtual memories, and persistent stores. They point out that exception handling will be a major component of the cost of their mechanisms.

2.2 Previous Approaches to Exception Handling.

Every operating system has some mechanism for addressing user-level exceptions. These have had varying degrees of success. In most cases designers handled exceptions in an *ad hoc* manner to serve the goals of their particular systems. We describe a few previous approaches to exception handling and we point out their strengths and weaknesses in meeting the requirements for our applications.

2.2.1 Unix Signals and Ptrace

In Unix [RT74] the user-level exception handling mechanism is called *signals*, a form of software interrupt [SPG91] that allows a process to install within it handlers for events that it causes or that otherwise affect it. The signals mechanism has many drawbacks from the perspective of our intended applications. Some of these are known problems on uniprocessors. It is possible to lose events. Currently, there are only two user-defined signals, thus making it difficult to introduce new exceptions in a consistent manner. Having the handler execute within the context of the affected process has the advantage of lowering trap-handling latency because there is no need to change processor or address space context, but decreases flexibility and safety. For example, there is no guarantee that the handler will be able to execute successfully in the case of severe problems. Having the handler execute within a single process requires maintaining several contexts for a single thread of control and can introduce naming problems. Finally, the signals mechanism does not support concepts such as multiple (system or lightweight) threads within a process, nor does it recognize multiple processors.

Unix has a separate user-level exception handling mechanism, called *Ptrace*, that corrects some of the deficiencies of signals. Used primarily as an interface to debugging tools, Ptrace provides much of the same functionality as signals, but with the addition of safety. The cost of using Ptrace, however, is much higher than that of signals and many of the original deficiencies with respect to our intended applications still remain.

2.2.2 Mach

The Mach exception handling mechanism [BGH+88] was designed specifically with multiprocessing in mind. The origins of PLATINUM in Mach are reflected in the reliance of the exception handling mechanism on message passing, but there are many significant differences between the systems. First, memory management exceptions associated with a memory object within a Mach task are not handled by the exception mechanism, but rather by a distinct set of primitives for supporting external pagers and other user-level virtual memory operations.

Another distinction is that the design of the Mach exception-handling mechanism is motivated specifically by its intended application to error handling and debugging. An exception is modeled as a form of remote procedure call. While this is adequate for simple cases, we shall see examples in which the RPC model is not appropriate. In particular, there are cases in our implementation of a remote debugging environment and in our proposed lightweight thread package in which the interrupted thread is resumed either by an entity other than the exception handler or at a location other than the one the thread was in when the event was reported.

In both the Mach and the PLATINUM mechanisms, there is a binding from each major kernel-defined object to a *port* to which exceptions associated with it are reported. In Mach, the kernel tries to make a distinction based on functionality when choosing between sending an exception message to a thread's exception port *versus* to the exception port for the task (address space) in which the thread lives. Exceptions that the designers believed should be processed by error handlers are reported to the thread exception ports while exceptions that are nominally for debugging purposes are reported to the address space exception port. Thus, all exceptions are classified statically at system design time into these two categories. This early, static binding becomes troublesome when we allow exceptions to have different semantics in different programs or even in different runs of the same program. Furthermore, since all events associated with debugging in an address space are sent to a single port, this design serializes the handling of debugging events, thus precluding our strategy of allocating one debugging agent per processor or per thread of control.

2.2.3 Medusa

Many of the ideas in the Mach and PLATINUM exception handling mechanism were pioneered in the Medusa operating system [Ous81]. The exceptions caused by a process (called an "activity") could be handled by several entities. Each exception is reported to an "internal" handler responsible for cleaning up the control structures of the target activity and, optionally, to an "external" handler responsible for cleaning up the state of shared objects. The internal handler can be within the target activity, a "buddy" activity within the same application, or a "parent" activity in another protection domain. Exceptions associated with a shared object are reported to *every* activity that has access to it. Inter-activity exception reporting was done using low-level kernel messages delivered to "flagboxes". The mechanism was extremely flexible in that exceptions were partitioned into classes and a separate handler could be dynamically specified for each class of exception for each object or activity.

From the perspective of our intended applications, some details of the implementation pose serious problems. Medusa had a very small kernel that routed all exceptions in the system through a single, centralized system activity (called a "utility") known as the *exception reporter* (also called the *exception manager* in [GSS87]), from which they are dispatched to the correct handler(s). In addition to serializing exception reporting, this centralization makes it difficult to experiment with hierarchical exception handling. The strategy of redundantly reporting exceptions to multiple handlers, especially for shared objects, is useful for providing very robust fault-tolerance, but it is a poor match for the kinds of applications of user-level exception handling that we mention above.

Like Unix, Medusa provided a separate utility (called MACE) specifically to handle exceptions associated with debugging and tracing.

2.3 Programming Language Support for Exceptions

Exception handling is also an issue in the design of a programming language and its runtime environment. Languages like ADA [Ich77], PL/I [Mac77] and Mesa [MMS79, GMS77] provide primitives for exception handling within the language. There has been, however, a strong debate on how useful these mechanisms are [Bla82] since they have an adverse impact on the simplicity and elegance of the language. Regardless of the outcome of this debate, it is our belief is that the runtime environment of a language will provide some means of cooperating with the operating system for exception handling even if the language does not have exception handling primitives *per se.*

3 An overview of the PLATINUM kernel interface.

In this discussion we focus on those aspects of PLATINUM that are directly relevant to exception handling. Description of the other aspects of PLATINUM can be found in [FC88, CF89, Cox91].

3.1 Programming Model

The fundamental abstractions supported by PLATINUM are the *thread*, the *memory object*, the *port*, and the *address space.*

A *memory object* is an abstraction of an ordered list of memory pages. A range of pages within a memory object may be bound to any contiguous page-aligned virtual address range of the same size. Memory objects are the natural unit of data- or code-sharing between address spaces.

A *thread* is a kernel-scheduled thread of control. At any time it is bound to a single processor. An explicit migration operation can move it to another location. It is constrained, however, to execute within a single address space. The kernel uses strict, preemptive, priority scheduling, with threads of equal priority using round-robin time slicing.

An *address space* is a list of bindings of memory objects and access rights to virtual address ranges. It defines the environment in which one or more threads may execute. The threads in a single address space may be distributed to multiple processors.

A *port* is a message queue that can have any number of senders and receivers [1]. Messages are untyped, variable-length arrays of zero or more bytes. Globally named, ports provide a communication medium usable by threads that do not share a common memory object. They also provide blocking synchronization.

[1] A message queue that allows multiple receivers is usually called a "mailbox". The use of "port" reveals the Mach ancestry of PLATINUM.

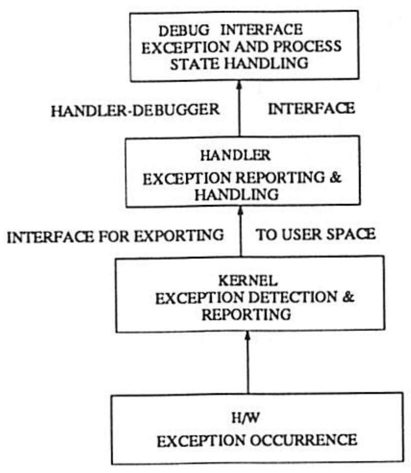


Figure 2: Layers of the exception handling mechanism

These objects all appear in a single flat global name space. Possession of a name conveys certain access rights to the named object [2]. Whenever an object is created, a corresponding *identifier port* is also created with it. An object is named by referring to its identifier port. Since ports and other objects have disjoint sets of operations, this introduces no ambiguity. When an object is destroyed, the kernel sends a message to the identifier port. This termination message conveys completion codes and error messages to a process, such as a shell, that is waiting for the completion of a thread or a user program.

Logical concurrency is realized through the use of multiple threads to implement a single application. True parallelism is realized by running those threads on multiple processors. Communication between threads can be based on either shared memory or message-passing *via* ports. A more restricted form of sharing is realized by mapping a memory object into multiple address spaces. The shared object can be accessed by all of the threads in those spaces, but the non-shared objects in each address space are protected from threads in other spaces.

4 Exception Handling in PLATINUM

Exception handling is divided between kernel and user-level modules. The kernel provides a single, general mechanism for exporting exceptions to user space. Policy issues are addressed entirely at the user level, with the shell and the runtime library defining a reasonable set of initial defaults. The mechanism is flexible enough, however, to allow user-defined modules to override any of the defaults.

4.1 Structure.

The PLATINUM exception handling facility is distributed across several layers of abstraction (see Figure 2) with well-defined protocols linking the components at each level. Hardware provides the lowest level of exception detection and reporting. Exceptional conditions can also be detected by software and inserted at several different points in the hierarchy.

The kernel exception handler is implemented in two sub-layers above the hardware. The lower sub-layer provides a "fast path" for handling memory management events associated with the software caching mechanism of Coherent Memory [CF89, Cox91]. The higher sub-layer deals with other kernel-handled events. The kernel exception handling layer associates each possible exception with some object, and each object names an *exception port* that is to be used as a destination for the

[2]Names can be passed in messages and through shared memory. In the PLATINUM research prototype, names are not encrypted and possession of a name confers all rights to the named object.

exception messages associated with that object. While some events are sent as messages to kernel-private ports associated with device-driver threads that execute within the kernel's address space, most are forwarded to user level through the exception port mechanism.

While the binding between an object and its identifier port is permanent, the binding between an object and its exception port is dynamic and can be changed at any time. Multiple objects can share a single exception handler by naming a common exception port. Since exception messages contain the name of (and thus convey access privileges to) the relevant object, this sharing puts no special burden on the shared exception handling service, but it does require that the user program trust the handler.

The kernel exports the following operations to manipulate exception ports and their bindings.

bind_exception_port(object_id, port_id) - installs the named port as the exception port of the named object.

get_exception_port(object_id) - returns the identifier of the object's exception port.

The third layer consists of a user-level exception handler. It is an ordinary thread that receives exception messages on an exception port. The exception message contains the identifier of the object that caused the exception, an exception number, and then, to save the cost of subsequent system calls that would otherwise be needed, an exception-specific variant part that contains information commonly needed by the handler. Because the relevant objects are always named in the message, there is no ambiguity as to which object is affected, and the handler is guaranteed the access rights that it needs.

Some of the kernel primitives commonly used by a handler to examine and manipulate objects are

suspend_thread(thread_id) - Suspend the thread with id **thread_id**. While provided primarily for user-level policies for scheduling kernel threads, **suspend** and **resume** are used extensively by the exception mechanism. In particular, the kernel suspends any thread causing an exception.

resume_thread(thread_id) - Makes a suspended thread eligible for scheduling. This places a non-blocked thread in the appropriate ready queue.

get_thread_state(thread_id, registers) - returns the register contents of a suspended thread **thread_id**. The register contents are also included in some exception messages.

set_thread_state(thread_id, registers) - sets processor state of the suspended thread **thread_id** to **registers**.

get_address_space(thread_id) - returns the identifier of the thread's address space.

read_range(address_space,source_addr,size,dest_addr) - reads **size** bytes starting at memory location **source_addr** from address space **address_space** and puts them in memory location **dest_addr** in caller's address space.

write_range(address_space,dest_addr,size,source_addr) - writes **size** bytes starting at memory location **source_addr** in the caller's address space to address space **address_space** at memory location **dest_addr**.

Kernel-level device drivers and user-level code communicate through ports that are specifically allocated for this purpose. Communication with these devices usually is not handled through the exception mechanism, though it is sometimes advantageous to redirect I/O and timer messages to a port that is read by an exception handler. Error conditions, breakpoint, and trace traps are associated with the executing thread. Memory exceptions can be associated with either the active address space or a memory object. If the address of the faulting memory operation is within the range to which a memory object is mapped, the exception is associated with that object. Otherwise, it is associated with the address space.

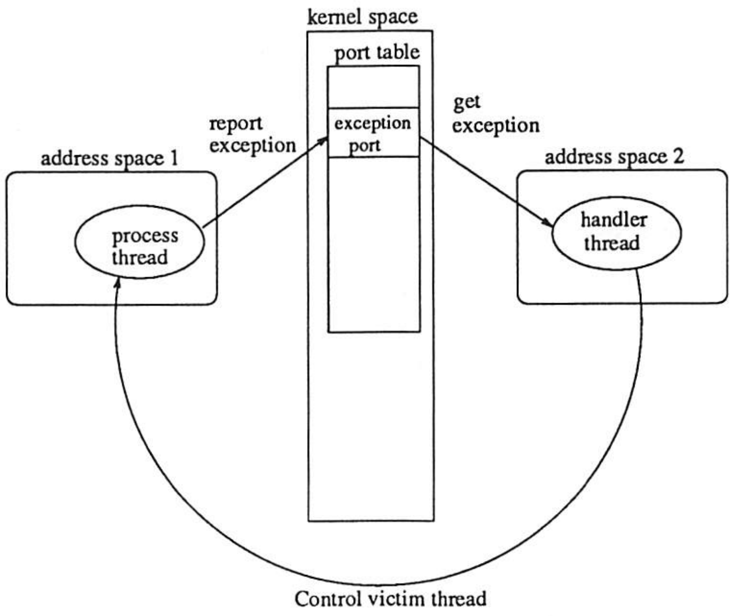


Figure 3: Process - Handler interaction

Since user-level exception handlers are ordinary threads, there are several techniques that can be used to modify the exception-handling environment in which a program executes. The only two parts of the mechanism that cannot be customized are the exception message format and the system calls that support exception handling.

One customization technique is to construct standard handlers that allow user programs to override default actions by specifying that certain exceptions should be forwarded to non-standard handlers. When used within the address space of the application, this technique can simulate Unix Signals or invoke other language-specific exception mechanisms. When used in a separate address space, this technique can yield the same functionality as Medusa's exception reporter.

A second customization technique is to use dynamic exception port binding to redirect exceptions for one or more objects to a replacement handler. These can be standard replacements, such as debugger agents, or they can be custom code. The replacement handler may even be an identical copy of the original in situations in which a program-wide handler is replaced with per-processor or per-thread handlers for performance reasons.

The third customization technique is to use dynamic exception port binding to direct exception messages to to a "filter thread" that intercepts a certain class of exception and forwards all unhandled exceptions to the original handler. This is a simple solution, but because it entails passing an extra message, this simplicity is obtained at the price of added overhead. It is the preferred option, however, when exception handling is being used simultaneously for multiple purposes such as attaching a debugger to a lightweight threads package,

In some cases, the user-level exception handler may cooperate with other user-level programs. For example, for remote debugging, we created a debugger host server that runs on a processor with an Ethernet interface. The host server manages communication, especially protocol translation, between the debugger agents that handle exceptions and a debugger (GDB [Sta87]) running on a remote workstation.

4.2 Defaults

Although the mechanism is flexible, very few programmers will want, or really need, to define their own schemes for exception handling. We therefore provide a reasonable default configuration.

A typical application program on PLATINUM consists of a single address space into which several

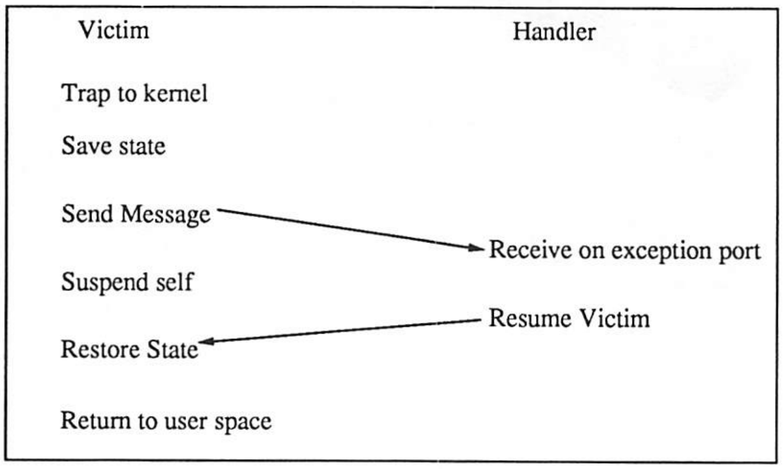


Figure 4: Exception path

memory objects are mapped and in which several threads run in parallel. I/O is handled via message passing to appropriate servers. The exceptions generated by this typical application can thus be reported to the exception ports bound to the address space, the threads, or the memory objects. The kernel, the simple Unix-style shell from which programs are run, and the initialization code of the run-time library cooperate to set up the default handling.

The kernel's job is the simplest. Whenever an address space is created, the kernel initializes the space to name a newly allocated port to be its exception port. When a thread is created, its exception port slot is initialized to name the exception port of the address space in which it is created. On creation, memory objects and ports are bound to system-wide standard exception handlers.

The shell and the run-time initialization routine cooperate to set up the handler. To run a program, the shell first creates an address space in which to run the program. It then loads the program by mapping the objects corresponding to the statically allocated text and data in the executable file. It also creates other memory objects for uninitialized data and stack space. It then creates and starts two threads. The *exception handler thread* begins to execute the default handler in the context of the shell's address space, immediately blocking in an attempt to receive a message on the address space exception port. The *root thread* of the application runs in the new address space. The shell then waits for the program to complete by blocking with a receive on the identity port of the root thread. If the user does nothing else, the overhead of setting up exception handling is thus the cost of creating one port and one exception-handler thread that spends its entire lifetime blocked waiting for an exception.

In the default scheme, installing an alternative handler for any one thread is a simple matter of rebinding its exception port. The default handler continues to try to receive messages on the exception port of the address space. Since the address space exception port is well known, it is a simple matter to de-install the alternative handler or to forward individual messages to the default handler.

5 Measuring and Enhancing Performance.

In computer systems, it is common to pay a performance penalty in exchange for the generality and flexibility that comes with a modular or layered system design. If the overhead of such a design is too great, the system will not be used, so it is important to pay attention to performance. In this section we discuss measurements of the performance of the PLATINUM exception handling mechanism. We also present the implementation strategies, similar to those in [BALL90], that we used to improve latency in those cases in which the full generality of the mechanism is not needed.

The measurements are based on a simple test application that consists of an address space in

	Save State	Suspend-Resume	Send Message	Total
Butterfly(unoptimized, 2 proc)	234μsec	801μsec	454μsec	1.480msec
Butterfly(unoptimized, 1 proc)	234μsec	881μsec	474μsec	1.597msec
Butterfly(optimized, 2 proc)	234μsec	811μsec	414μsec	1.456msec
Butterfly(optimized, 1 proc)	234μsec	690μsec	232μsec	1.160msec
Sun3/50 (ptrace)	-	-	-	1.760msec
Sun3/260 (ptrace)	-	-	-	1.590msec
Sun4 [3] (ptrace)	-	-	-	0.680msec
Butterfly(no context switch)	234μsec	-	454μsec	0.703msec
Sun3/50 (catch signal)	-	-	-	0.813msec
Sun3/260 (catch signal)	-	-	-	0.757msec
Sun4 (catch signal)	-	-	-	0.214msec

Table 1: Performance measurements for exception handling [4].

which there is a thread with an inner loop containing an integer division instruction. The control experiment consists of measuring the time necessary to execute the loop. To measure the cost of exception handling, we set the divisor to zero and we time the loop using an exception handler that immediately resumes the trapping thread as soon as the exception is reported. The difference in measured execution times thus includes all of the overhead incurred in handling the exception. The actions taken are illustrated in Figure 4. We divide the cost of servicing an exception into three constituent subcosts: the costs associated with entering and leaving the kernel, saving and restoring the target thread's state; the time to send the message to the handler, including context switch and scheduling costs; and the cost of the context switches involved in suspending and resuming the target thread. The component costs are measured with modified versions of the kernel and the test application that selectively omit some of the actions.

To define a base case, we measured the average latency of exception handling in the case in which the target and handler are each the highest priority user threads running on two different processors. The measured latency is 1.48 milliseconds per exception. Running the handler on the same processor as the target increases the latency to 1.6 milliseconds per exception. The difference in execution time is due to a loss of parallelism between those parts of the protocol that can be executed concurrently. This latter figure is thus a better indication of the total overhead of the operation.

More than half of the exception service time is spent in suspending and resuming the faulty thread. A large part of this is the cost of switching contexts, especially of changing address spaces. This latter cost is unavoidable when the target and handler are in different address spaces or when they are run on different processors, whether they share an address space or not. On the other hand, it is a simple matter to check whether the old and new threads at a context switch are in the same address space and to take a "fast path" that avoids address space operations. We added this optimization to all context switches performed by PLATINUM. Performing the check adds an average of 11 microseconds to the cost of a context switch if it is necessary to change. The average net benefit when the threads are in the same address space is about 380 microseconds. The benefit of this optimization was not realized by the simple test program when it was run on two processors because it contained only two threads, one per processor. When either thread is blocked or suspended, the kernel runs an idle thread in a different address space.

If there is a thread blocked on a port waiting for a message to arrive, the cost of sending a message can be improved by copying the body of the message directly between user-level buffers. This modification yielded only a modest speed improvement of 40 microseconds per exception, since most exception messages are relatively small.

[3]The Sun 4 is a SPARCstation 1.

[4]The difference between the total cost and the constituents is ascribable to roundoff errors and to other minor actions taken in the protocol

The effects of these optimizations are summarized in Table 1. We also compare the latency of PLATINUM exception handling with the cost of related operations on several uniprocessor systems using Unix signals and Ptrace. For each machine we performed three sets of measurements. The control runs were as on PLATINUM. We then measured the cost of exception handling using signals to catch the traps. Finally, we measured the latency of using Ptrace to catch the traps in another process.

In terms of processor speed, the Sun 3/50 workstation is the closest of the three systems measured to a Butterfly processor operating with code and data in its local memory [5]. Both are based on Motorola 68020 processors running at similar clock speeds (approximately 16MHz) and neither has a data cache. The measurements show that the unoptimized PLATINUM implementation is slightly faster than the Sun using Ptrace, but that when the exception is handled by a thread in the target address space the optimized PLATINUM mechanism is about 40% slower than using signals. Thus, when the full generality and safety are required, there appears to be no performance penalty in using the PLATINUM mechanism, but there is a considerable cost difference if the exception can be handled in the same address space and on the same processor.

A major part of the expense continues to be the context switching and scheduling overhead incurred by using a separate kernel-scheduled thread of control as the exception handler. One way of avoiding this is to handle the exception within the same kernel-scheduled entity that incurred the trap. The basic flavor and flexibility of the PLATINUM exception protocol could be retained if the kernel interface supported abstractions such as First-Class User-Level Threads [MSLM91] or Scheduler Activations [ABLL91] that would block only one lightweight thread on a trap. To test this, we hand-coded an experiment in which the target thread contained two lightweight threads. On a trap by lightweight thread, the kernel sends the exception message to the exception port and then resumes the other lightweight thread with no context switches. This lightweight thread receives the exception message from the port and uses its contents to resume the original computation. This eliminates the context switch and scheduling overhead, but the cost of posting the message and of the system call to receive the message is still a very large part of the total. The results of this experiment are reported in Table 1 as "Butterfly (no context switch)". Further improvements in the time to handle an exception would require abandoning our general model in favor of a mechanism that does not use PLATINUM ports and messages to transfer data.

It is difficult and dangerous to try to evaluate the relative performance of an operating system mechanism, either the abstraction or its implementation, by comparing its execution time with measurements of other abstractions on other architectures. There are too many uncontrolled architectural variables to make a definitive comparison and the standard measures of system performance for application programs are notoriously optimistic and inaccurate predictors of operating system performance [ALBL91]. Nevertheless, the results of our measurements indicate that, when operating on a single Butterfly node, the PLATINUM mechanisms are comparable to or faster than the Unix operations with similar functionality executing on machines with equal or greater power. Furthermore, our multiple processor timing numbers are worse than the single processor figures only to the extent that uniprocessor optimizations are inappropriate. Unfortunately, there appear to be no recent measurements in the literature of similar operations on multiprocessor systems.

6 Examples Illustrating the Generality of the Mechanism.

In this section we support our assertions about the generality and flexibility of the exception handling mechanism by presenting examples of its use in several diverse settings.

6.1 Debugger Interface.

In Figure 5 we illustrate in more detail how the exception handling mechanism is used to mediate between parallel applications on PLATINUM and an instance of the GNU Debugger [Sta87] running

[5]The other systems were measured in order to provide an indication of how the costs might change with improved technology.

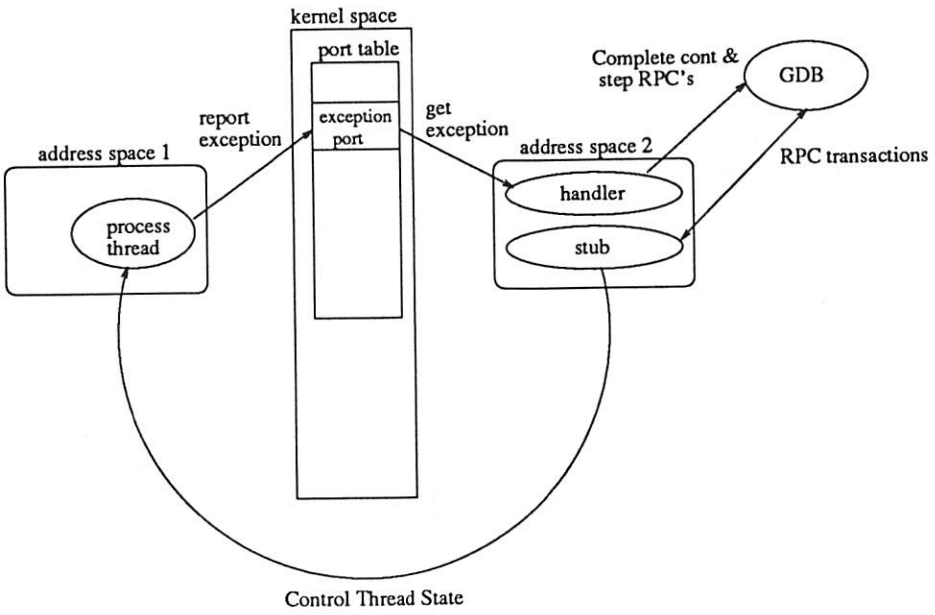


Figure 5: Debugging scheme

on a remote workstation. All of the exception handler/debugging agent threads share an address space with the debugging host server thread. The agents run on the same processors as their target threads and the host server runs on the processor with the Ethernet controller. The handlers are normally blocked waiting for messages from their respective target threads. The host server can block only waiting for GDB to initiate an RPC transaction. Operations requested through GDB are performed by the host server. Events occurring in the target threads are reported back to GDB by the agents. For example, the user could type control-C to interrupt the "current thread". The request is sent to the host server which suspends the thread, sends a software-exception message to the thread's exception port, and then blocks waiting for the next request. This unblocks the correct handler to complete the request and reply to GDB. The protocol for communicating with GDB is encapsulated in a set of functions that are logically part of the host server program but are accessible for execution from the handler threads. These functions also encapsulate the synchronization necessary to multiplex the messages back to GDB. Other requests, such as setting breakpoints or reading and writing memory are handled entirely by the server thread.

The one debugging primitive in which performance is critical is the handling of conditional breakpoints. In our prototype these are handled by an instance of GDB running on a remote workstation, so the added communication cost is high. Hand-coded experiments, however, indicate that by migrating the function of testing the condition to the debugging agent we can improve the latency of each test by about two orders of magnitude when debugging a single thread. For debugging multiple threads the advantages will be much greater since the condition testing will be done in parallel. The integration of this technique into a production quality debugger as well as adding other primitives deemed useful or necessary for debugging multithreaded programs [BGH+88] appears to be straightforward and is left for future work.

6.2 Supporting User-Level Threads.

Several groups [ABLL91, MSLM91] have pointed out that efficient kernel support can be crucial to the performance of user-level thread packages. Such packages can take advantage of our optimized intra-address-space notification path by using an exception handler as the user-level thread scheduler. Figure 6 illustrates such a user-level thread package. There are two kernel-scheduled threads allocated on each processor used by the application. One of these serves as the virtual processor

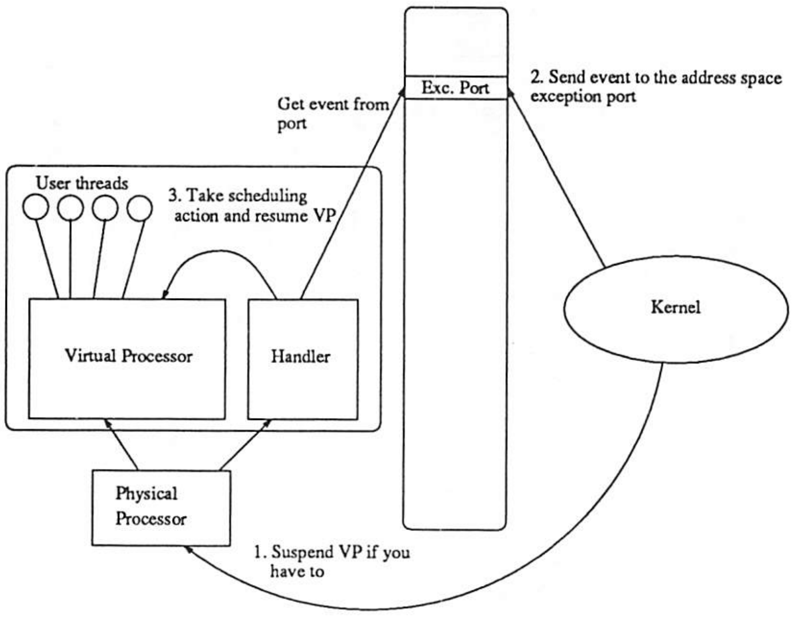

Exc. Port
2. Send event to the address space exception port
Get event from port
User threads
3. Take scheduling action and resume VP
Kernel
Virtual Processor
Handler
Physical Processor
1. Suspend VP if you have to

Figure 6: Lightweight thread scheduling

on which lightweight threads run. The exception handler thread is responsible for the event-driven scheduling actions affecting the lightweight threads on the virtual processor thread.

The kernel sends all upcalls that affect scheduling on this processor to the exception port of the virtual processor. Messages from I/O server threads and from the interval timer device used for time slicing are also sent to the exception port. This necessitates defining several new software interrupts, such as "thread would block" and "I/O completion", that the kernel can raise when appropriate. The handler-scheduler would thus be event-driven by these upcalls. Each exception message causes the handler-scheduler to run and perform an action such as choosing a new lightweight thread to run or migrating a lightweight thread to or from the virtual processor. Voluntary context switches between lightweight threads can be handled entirely in user space, thus very efficiently, by allowing the virtual processor thread to access the scheduling functions directly. This proposed organization can be used to simulate efficiently several of the user-level thread management abstractions in the literature. Simulating the "two minute warning" preemption avoidance mechanisms of [MSLM91] is straightforward, with the handler-scheduler performing exactly the same actions as are performed in the original implementation. Alternatively, the handler-scheduler can be set up to emulate the upcall mechanism of the Anderson *et al.* Scheduler Activations [ABLL91]. In this case, preemption recovery would be performed by sending an appropriately encoded message to the suspended handler-scheduler's exception port. Given that the states of all of the suspended threads are available to the exception handling mechanism, the current state of the preempted lightweight thread is also available and can be run to completion on another node.

We note that, in the current implementation, the overhead imposed by all of these operations is under 1.2 milliseconds, and that they can be implemented by extending the kernel to define new exceptions, but without radically changing the semantics of the basic objects in the system. Our performance measurements indicate that the cost per upcall could be brought down to about 700 microseconds on the Butterfly by changing the semantics of the kernel interface to include an abstraction like Scheduler Activations, while keeping most of our current implementation of the kernel and of the exception reporting mechanism.

7 Conclusions

When designing the exception handling mechanism, we set a number of ambitious goals for a single mechanism to achieve simultaneously. These included conventional exception handling, single-threaded and multi-threaded program debugging, lightweight thread support, and external paging. Our experiences with the PLATINUM exception handling mechanism have confirmed that we have achieved our goal of providing a simple, flexible, and efficient mechanism for user-level exception handling. We have shown how the mechanism can be used for error handling by creating a handler thread and allocating an exception port for the handler-thread communication. We have shown how to use the same model to interface a debugger, and we have commented on benefits that multithreaded debuggers can derive from our approach. We have also demonstrated how the mechanism can be used to provide kernel support for user-level threads at a very low cost using our optimized intra-address-space notification path. Finally, we have noted that the mechanism's generality of binding exception ports for any object in the system can allow for use-level virtual memory management. We believe that it is worth reimplementing our model for exception handling using new techniques for kernel and server structuring like continuation passing [DBRD91] and memory mapped message passing [BALL90] as well as exploring the possible performance benefits that our mechanism will yield under these different models of communication and thread management.

8 Acknowledgements

This work owes a lot to the help of Alan Cox, whose innovative comments and suggestions based on his intimate knowledge of every detail of PLATINUM proved invaluable.

References

[ABLL91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium in Operating System Principles*, pages 95–109, 1991.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth ASPLOS*, pages 96–107, Santa Clara, CA, April 1991.

[ALBL91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth ASPLOS*, pages 108–122, Santa Clara, CA, April 1991.

[BALL90] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[BGH+88] D. L. Black, D. B. Golub, K. Hauth, A. Tevanian, and R. Sanzi. The Mach Exception Handling Facility. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, pages 45–56, Madison, Wisconsin, May 1988. Association for Computing Machinery.

[Bla82] A. R. Black. *Exception Handling: The Case Against.* PhD thesis, Oxford University, 1982.

[CF89] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 32–44, Litchfield Park, AZ, December 1989.

[Cox91] A. L. Cox. *The Implementation and Evaluation of a Coherent Memory Abstraction for NUMA Multiprocessors.* PhD thesis, University of Rochester, 1991.

[DBRD91] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 122–36. Association for Computing Machinery SIGOPS, October 1991.

[Els88] I. J. P. Elshoff. A distributed debugger for Amoeba. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, pages 1–10, Madison, Wisconsin, May 1988. Association for Computing Machinery.

[FC88] R. J. Fowler and A. Cox. An overview of PLATINUM: A platform for investigating non-uniform memory. Technical Report TR-262, Computer Science Department, University of Rochester, November 1988.

[FLMC88] R. J. Fowler, T. J. LeBlanc, and J. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distribute Debugging*, pages 163–173, Madison, Wisconsin, May 1988. Association for Computing Machinery.

[GMS77] C. M. Geschke, J. H. Morris, and E. H. Satterthwaite. Early experiences with Mesa. *Comunications of the ACM*, 20(8):540–553, 1977.

[GSS87] Edward F. Gehringer, Daniel P. Siewiorek, and Zary Segall. *Parallel Processing: The Cm* Experience.* Digital Press, Bedford, MA, 1987.

[Ich77] J. D. Ichbiah. Rationale for the design of the ADA programming language. *SIGPLAN Notices Vol. 20*, pages 500–503, 1977.

[LE80] Henry M. Levy and Richard H. Eckhouse, Jr. *Computer Programming and Architecture, The VAX-11.* Digital Press, Bedford, MA, 1980.

[Mac77] M. D. MacLaren. Exception handling in PL/I. Technical report, Digital Equipment Corporation, Maynard, Mass., 1977.

[MMS79] J. G. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual Version 5.0.* Xerox Palo Alto Research Center, Systems Development Department, 1979.

[MSLM91] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 110–121. Association for Computing Machinery SIGOPS, October 1991.

[Mul87] Sape J. Mulleneder. *The Amoeba distributed operating system : selected papers, 1984-1987.* Centrum voor Wiskunde en Informatica, 1987.

[Ous81] J. K. Ousterhout. *Medusa: a distributed operating system.* UMI Research Press, 1981.

[Ras86] R. Rashid. From RIG to Accent to Mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society 1986 Fall Joint Computer Conference*, November 1986.

[RT74] D.M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[RTY+88] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[SPG91] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison Wesley, 1991.

[Sta87] R. M. Stallman. *GDB Manual: The GNU source-level Debugger.* Free Software Foundation, 1987.

Experiences with Accommodating Heterogeneity in a Large Scale Telecommunications Infrastructure

John R. Nicol, C. Thomas Wilkes,
Richard D. Edmiston, Joseph C. Fitzgerald

GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
e-mail: [jrnicol, ctwilkes]@gte.com

Abstract

As one of the largest local exchange carriers in the world, GTE supports a vast infrastructure of resources including numerous computers, switches, databases, networks, operating systems, protocols, programming languages, and applications. Due to huge financial investments in infrastructure, telecommunications organizations such as GTE are strongly motivated to gain as much leverage from available resources as possible. In addition, the sheer scale of telecommunications infrastructures is creating growing system maintenance and evolution problems.

The Distributed Computing Systems (DCS) Project was established at GTE Laboratories to address these issues through studies of the applicability of modern distributed computing and software engineering techniques to the telecommunications industry. The overriding complicating factor in the telecommunications environment is the massive heterogeneity of the present infrastructure. Therefore, we view the attainment of *interoperability* as a key goal in our work, and believe an object-based approach using "guest layer" technology presents a uniform and vital framework for achieving this goal.

In this paper, we discuss the problems presented by heterogeneity in the telecommunications environment; an initial approach to addressing some of these problems in the context of an object-based distributed system testbed; two telecommunications-related applications developed on this testbed as demonstrations of how to achieve interoperability between existing components, with minimal changes; a summary of our experience with the testbed; and conclusions drawn from this experience.

1. Introduction

For many decades, large organizations have faced the difficult problem of how to obtain maximum benefit from available resources. Since the late 1950s, the rapid evolution and application of Information Technology has been central to addressing this problem. For example, during the 1960s and 1970s database technology emerged, promoting the controlled sharing of centrally held corporate data. In addition to reducing resource overheads (through the elimination of vast amounts of hardware, software, and data redundancy), databases became able to present their users with conceptually simple models of the data available to them, while taking care of issues such as the prevention of unauthorized access to sensitive data, the prevention of data inconsistency resulting from conflicting concurrent accesses or system failure, and the preservation of the integrity of data held in the system. Although centralized databases have been very successful, their emphasis has been to gain maximum leverage from *data* resources.

The advent of distributed systems technologies in the late 1970s is now facilitating treatment of the more general problem of maximizing the use of physical resources such as devices (*e.g.*, processors, memory, switches, printers), networks (*e.g.*, LANs, MANs, WANs), and abstract resources (*e.g.*, programs, processes, objects, telephone calls) in addition to data in the traditional database sense. As well as increasing the leverage possible from available resources, distributed systems technologies provide the means to realize other important advantages to large organizations. These include the potential to construct systems exhibiting improved levels of fault tolerance, availability, extensibility, and performance [Nicol88].

The Distributed Computing Systems (DCS) Project was established at GTE Laboratories to address these issues through studies of the applicability of modern distributed computing and software engineering techniques to the telecommunications industry. The overriding complicating factor in the telecommunications environment is the massive heterogeneity of the present infrastructure. Therefore, we view the attainment of *interoperability* as a key goal in our work, and believe an object-based approach using "guest layer" technology presents a uniform and vital framework for achieving this goal. Furthermore, in order to make as much use as possible of the existing infrastructure, we desire interoperation among already-existing resources (*e.g.*, database systems, switch interface software) with minimal changes to those resources.

This paper is organized as follows. In Section 2, we discuss the nature of heterogeneity in the telecommunications environment and the problems it presents. We describe, in Section 3, our initial approach to addressing some of these problems in the context of an object-based distributed system testbed. In Section 4, we discuss two sample telecommunications-related applications, developed on this testbed as vehicles used to examine more closely some of the problems discussed in Section 2, and potential solutions to these problems. In Section 5, we summarize our experiences with the testbed and, finally, we present conclusions and future directions of our research in Section 6.

2. Heterogeneity in the Telecommunications Environment

There has been considerable interest within the academic community in problems relating to heterogeneity. This interest has been fueled to some extent through the practical need to deal with campus-wide computing environments containing a mix of machine architectures and operating systems, for example the Heterogeneous Computing System Project at the University of Washington [Notkin88] and Project Andrew at Carnegie-Mellon University [Satyanarayanan90].

Two essential differences between the environment in which such academic research efforts have taken place, and the environment typical of the telecommunications infrastructure, are those of *size* and *diversity*. For example, Project Andrew was considered extremely ambitious in setting out to accommodate up to 10,000 computing elements and only a few processor types. The scale of the telecommunications environment, however, is closer to that of the entire Internet: geographical dispersal possibly spanning a continent; dozens of different operating systems, switch and processor types; thousands of existing application programs; and at least an order of magnitude greater number of processing elements.

Another critical distinction between large-scale academic and industrial environments is that evolution in the latter is far more constrained; it is infeasible to replace large portions of the telecommunications infrastructure at one fell swoop for several reasons. The extremity of the financial investment required alone makes such action prohibitive. Furthermore, the scale of the system implies that the presence of bugs, for example following the deployment of new services, can have drastic financial consequences. Finally, the customer base demands continuous, uninterrupted service, thus discouraging wholesale replacement of system components.

Despite the infeasibility of this wholesale approach to evolution, the telecommunications environment is under constant pressure to change. Two major forces behind this are the need to remain competitive through the enhancement of subscriber services, and the need to integrate new technologies into the existing infrastructure. In practice therefore, significant changes do occur, albeit usually over extended timescales. For example, the migration from analog to digital switching technologies remains underway after more than a decade.

Due to the magnitude of the problems being addressed, and the difficulties of experimenting with solutions to these in the field, we adopted the approach of modelling the characteristics of the telecommunications environment via an experimental testbed. Although smaller and simpler than the real-world environment, it is our intention to use the testbed to model real-world considerations. Thus, decisions made and approaches devised for the testbed are intended to reflect those that would be made in the field. The ideal distributed systems testbed in a telecommunications setting would be one which can be employed as a platform for developing a unified approach to the continued support, maintenance, and evolution of advanced telecommunications applications and supporting software. Key requirements are to model and support:-

- operation in a highly heterogeneous environment;
- the porting of software to new systems;
- the facilitation of continued support of existing investments in systems and applications software and data; and
- the development of new and well-engineered software, while better integrating existing resources to exploit more fully the potential of distributed processing.

An important decision early in the project was the basic architecture of our testbed.

Practically all distributed operating systems developed to date can be readily categorized as either a *host system* or *guest system*. The category from which we would select our basic testbed system was central to our decision-making process, since such a decision essentially trades off crucial issues such as portability and user-impact against others such as performance and flexibility.

A host system comprises software that directly manipulates the resources of that system. Most centralized operating systems are host systems. *Distributed* host systems generally consist of a host kernel on each node of the network; the kernels interoperate to achieve global distributed control.

A guest system is usually characterized as a layer of software that extends an underlying host operating system (usually UNIX[1]) into a distributed setting. The earliest examples of guest systems essentially provided little more than a distributed file-system. More recently, guest systems capable of realizing more of the benefits of distributed computing have emerged (see Section 3).

More detailed treatment of host vs. guest system issues can be found in [Nicol86].

Given the characteristics of the telecommunications environment we are modelling, a distributed host system approach to our testbed would suffer from a number of major drawbacks. For example, substantial effort is required to design and implement a host kernel, and to port it onto new hardware platforms. Worse still, the introduction of a host system into the telecommunications environment would have a major disruptive impact (*e.g.*, by requiring programmers and users to adapt to a new environment, and by necessitating the modification of many existing applications).[2]

Advantages of adopting a guest system approach for our purposes are: it is generally much easier to port a guest layer onto new host operating system platforms than a native host kernel onto new hardware platforms; and the introduction of a guest layer need not disrupt users of the host system in any way. Indeed since a guest system layer essentially extends the functionality of the underlying host operating system, users are able to take advantage of both the host and guest system worlds.

Given our environment, a natural choice was thus to adopt the guest system approach. Due to the degree of effort required to develop a new guest system, and the availability of existing guest systems for most of the platforms of interest to us, we decided to select and adapt a third-party guest system. Among the candidates considered, BBN's Cronus [Schantz86, Vinter89] came closest to satisfying our requirements. In the following section, we present a brief technical overview of Cronus and discuss how it can be adapted to the telecommunications context.

3. An Experimental Testbed for Addressing Heterogeneity Issues

Our choice of Cronus to form the basis of a distributed systems testbed was strongly influenced by its designers' commitment to the following goals:-

- providing an effective, general-purpose, distributed computing environment composed of hosts featuring differing hardware and operating systems;
- supporting an integrated computing environment consisting of both the host operating system environment and the Cronus environment; and
- providing comprehensive support for the development of large scale distributed applications, with a focus on ease of use.

1 UNIX is a registered trademark of AT&T.

2 It should be noted, however, that many host kernels support an interface syntactically and semantically compatible with one or more of the leading flavors of UNIX. Though this may ease the impact of adopting a host system in a UNIX-oriented environment, it is unlikely to console users of VMS, MVS, MS-DOS, *etc.*

Cronus is also one of the most mature systems available, with over 75 man years of effort invested in its development since 1981. The system has been installed on several hundred nodes at over 30 sites, and a healthy number of non-trivial Cronus-based distributed applications have been deployed in the field. Cronus is supported currently on over ten widely-differing architectures (with others supported by prototype ports) under several different host operating systems. In this section, we present a brief technical overview of the system.

Cronus provides a set of services and communication layers on top of a host operating system. The system supports heterogeneity by providing a circumventible layer of abstraction between applications and the underlying host system. As a result, applications have access to a coherent and uniform distributed system interface independent of the supporting host systems, while retaining access to the usual resources and services made available by the host system interfaces.

The Cronus kernel is layered directly on top of the host systems, the primary purpose being to transmit operation invocations from clients to object managers in a host-independent way. The kernel also implements the basic abstractions for processes and objects, as well as providing facilities for monitoring and controlling other Cronus activity on the host. The kernel itself is implemented as a host system process executing in the user space of each host forming part of a Cronus cluster. Above the Cronus kernel reside *object managers*, which implement services, and *clients*, which make use of the services provided by the managers. (Naturally, a manager can take on the role of a client if it makes use of the services of another manager.) A high-level architectural view of Cronus is shown in Figure 1.

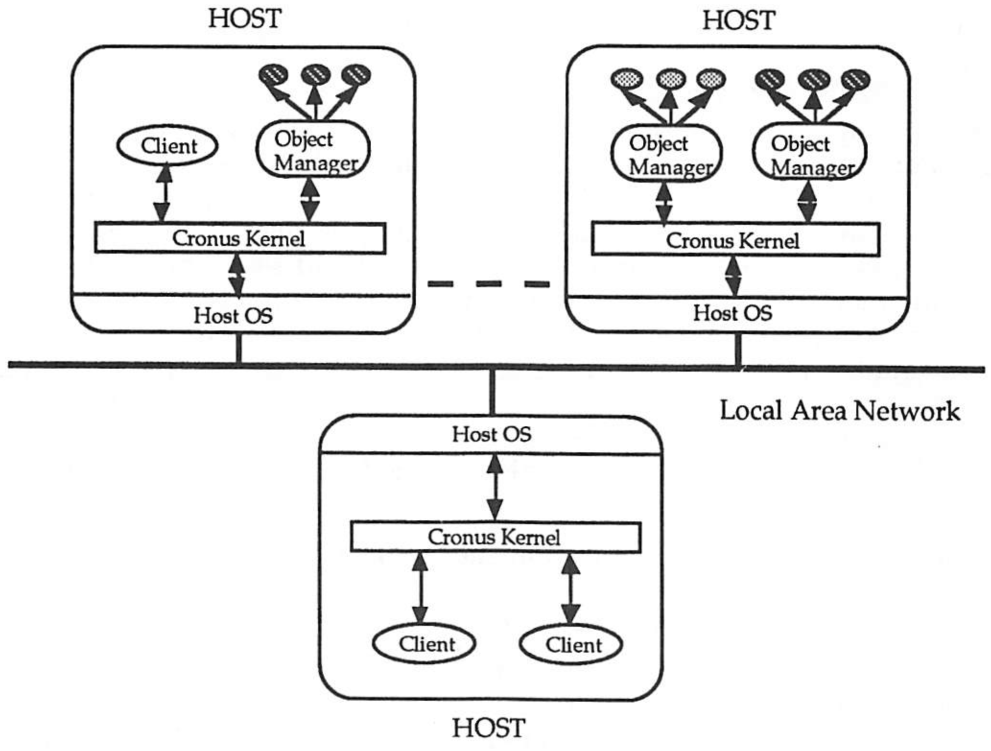


Figure 1: Cronus Architecture (adapted with courtesy of BBN)

Cronus includes a number of base object managers, including a name service, type definition service, file service, database access service, *etc.* It also includes a rich programming toolkit for expediting the development of new object managers (potentially building on existing managers via inheritance). New object managers are first-class in the sense that they are not distinguished from the base managers by Cronus. This feature allows one to adapt Cronus to fulfill new application requirements, in our case those of the telecommunications domain.

Examples of new object managers illustrating this adaptation process are discussed in the following section.

4. Applications of the Testbed

The initial focus of our work has been on adapting the Cronus-based testbed to the telecommunications environment. Our chief aims were to explore:-

- the re-engineering of existing (predominantly centralized) applications to operate in a distributed, heterogeneous environment;
- the enhancement of such applications to take advantage of fault tolerance, improved performance, and other advantages of distributed computing; and
- novel telecommunications-related applications.

Our desire for the testbed to reflect the considerations of the field environment imposed certain constraints, chief amongst which was the need to avoid modifications of existing code.

Towards these ends, we have developed several sample applications using the testbed. The remainder of this section describes two of these applications: an Enhanced 911 Service Prototype and a Video Object Manipulation Service.

4.1 Enhanced 911 Service

Our first application represents an example of the enhancement of an existing telecommunications application. The Enhanced 911 (E911) Service is a Local Exchange Carrier service, providing network access and subscriber information to public safety agencies. Any subscriber can initiate use of the service by dialing 911. The service is beneficial as it provides a universal and easily recallable number for dialing in emergency situations, quick routing, and continuous service at the *Public Safety Answering Point* (PSAP). In addition to the voice connection between the PSAP operator and the 911 caller, the PSAP operator is supplied with the phone line subscriber's name, address, telephone number, emergency information, location map, *etc*. The subscriber information is retrieved from a regional database, and displayed immediately on the terminal associated with the operator who answers a particular 911 call.

Switch control in this application has been addressed through the use of a previously existing external switch interface. This interface implements a set of primitives reflecting the capabilities of the exchange network.

In an earlier, less sophisticated, implementation of an E911 prototype, the application software connects a caller dialing 911 to a PSAP operator who assists with the emergency. Since all PSAP operators could be busy when a caller dials 911, it is not always possible to service a 911 call immediately. Under such circumstances, the application software gives the first such caller a recorded announcement, connecting the caller to a PSAP operator as soon as one becomes available. In addition, the number of announcement ports is limited; when there is neither a free PSAP operator nor a free announcement port for a new 911 caller, the application software gives the caller ringback signal to indicate the call has been received.[3] When a PSAP operator becomes free, the application software connects the free operator to the 911 caller who has listened to the announcement the longest. When an announcement port becomes free, the software connects it to the 911 caller who has listened to ringback the longest. The application software must maintain the queues (*i.e.*, PSAP operator queue, announcement queue, and ringback queue), and invoke primitives executed at the switch.

We employed the DCS testbed technology to integrate the existing E911 software with commercial database systems (e.g., Oracle[4] running under DEC VMS[5]). In addition to the aims stated at the beginning of section 4, we wished to address issues concerned with ease-of-construction from "active" applications. This application was developed in the context of the heterogeneous, distributed environment illustrated in Figure 2. The architecture of this environment reflects the "intelligent network" trend of off-loading functionality and control from the switch [Robrock91].

Two client programs (providing the retrieval and update capability) and a database server (providing the subscriber information) were developed to demonstrate the E911 service in this environment:-

- The E911 subscriber database manager was built to provide for the storage and manipulation of subscriber information (including switch port ID, name, address, telephone number, *etc*). This database manager was replicated on two heterogeneous hosts (a DEC VAX 6410 running VMS, and a Sun 3/60 running SunOS[6]) to provide high availability.
- The two client programs (running on Sun SPARC machines under SunOS) were written to enable remote access to subscriber state (object instances of the subscriber object manager) through the invocation of the operations supported by the subscriber database manager (*e.g., find, terminate,* and *create*). To demonstrate the distributed computing capability using the DCS testbed, the first client program provides the capabilities of remotely creating a new subscriber record and terminating an existing subscriber record. The other client program interfaces with the driver interface developed earlier, and accesses the database manager remotely to look-up the subscriber's details before subsequently displaying these on the data terminal associated with a PSAP. The client programs can manipulate the database concurrently.

In addition to Oracle, the E911 object manager has been readily extended to make use of another instance of a commercial database (Sybase[7]). This extension mainly involved the modification of the E911 client programs to access the subscriber tables created in Oracle and Sybase databases.

---

3 The logic behind this is that the caller may hang up if a busy signal is returned in response to his call.

4 Oracle is a trademark of Oracle Incorporated.

5 DEC, VAX and VMS are trademarks of Digital Equipment Corporation.

6 Sun and SunOS are trademarks of Sun Microsystems.

7 Sybase is a trademark of Sybase Incorporated.

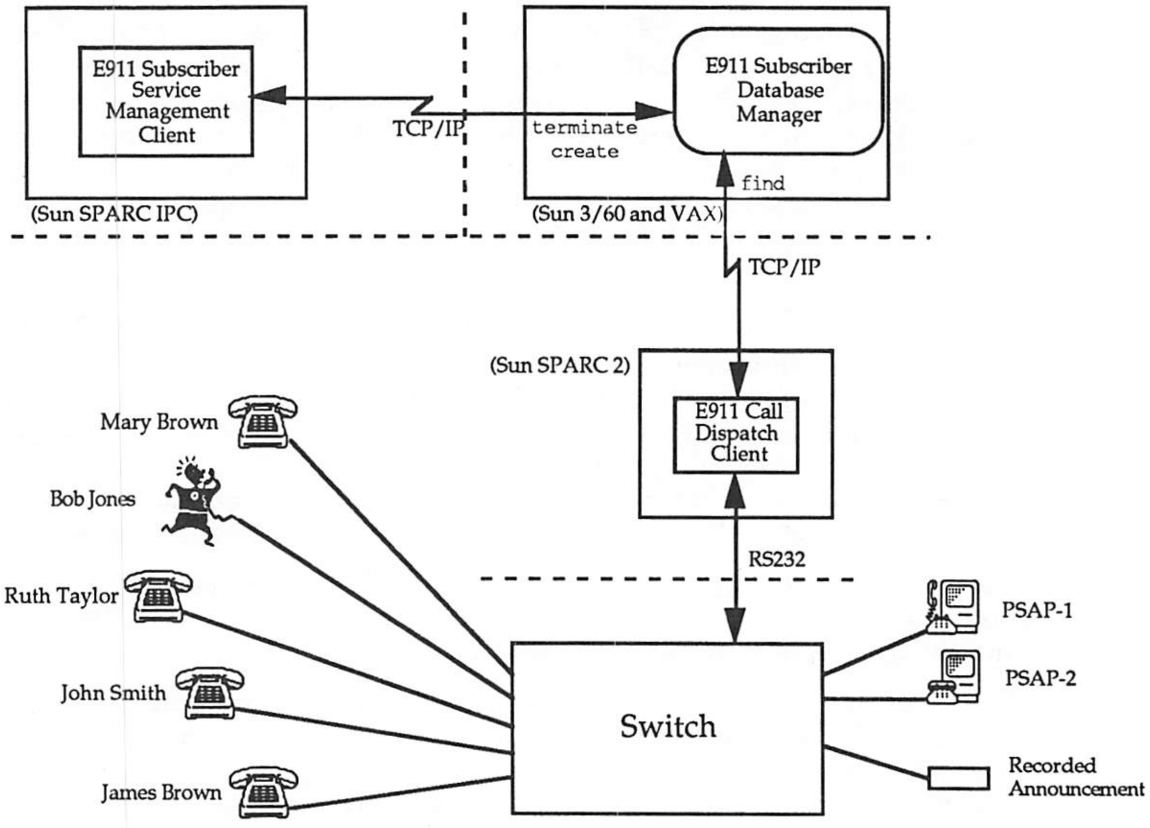


Figure 2: E911 Execution Environment

4.2 Video Object Manipulation Service

Our second application represents an example of a "novel" telecommunications application. The Video Object Manipulation (VOM) Service application described in this section was developed to enrich the *Athena Muse* environment [Hodges89], a powerful construction kit for multimedia applications.

Currently Muse includes a video server, *Galatea* [Appelbaum90], to manage video playback and switching devices distributed over computer and CATV networks. However, Galatea provides a very low-level abstraction over physical video devices, much like an operating system device driver. Galatea lacks support for high-level naming mechanisms, such as would be required for maintaining video archives. As such, Muse applications using Galatea typically include encoded knowledge of higher-level entities such as video segments (analogous to video files) in the form of hardwired offsets of positions of video frames on a given volume. Consequently, the use of Galatea suffers from a number of drawbacks:-

- its low-level nature makes it awkward to use and thus error-prone;
- it is inflexible—if frames on disk become displaced, (Muse) video applications require modification followed by re-compilation;
- the use of frame offsets in application code damages its readability; and
- convenient reuse of existing video abstractions is not possible.

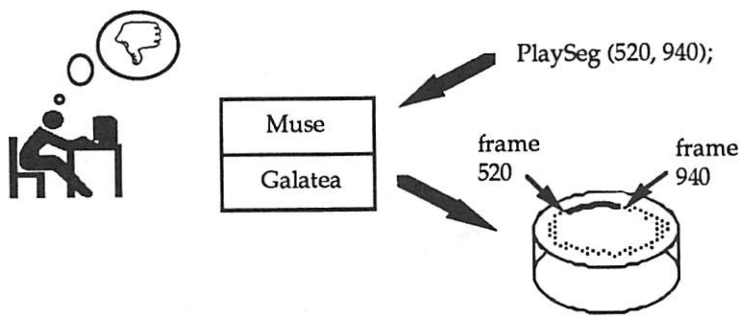


Figure 3: Muse Programming Prior to Introduction of Video Object Manipulation Service

Figure 3 shows a logical perspective of the use of Galatea from the viewpoint of a Muse applications programmer, in particular the use of explicit references to the positions of video frames stored on disk.

A major role of the VOM Service described here is similar to that of the file-system of a conventional operating system, *i.e.*, to provide higher-level naming abstractions over the low-level entities maintained by storage device controllers, thus offering a more flexible and sharable name service.

The VOM Service developed using the DCS testbed provides two abstractions over video frames: *video segments* and *video sequences.*

A video segment is a logically atomic, named series of video frames.[8] The video segment manager maps a user-oriented name of a video segment onto a descriptor containing lower-level video information (required by Galatea for the manipulation of the associated video frames). Information held in a video segment descriptor includes the name of the volume storing the associated video frames, the start and end frame offsets of the segment, the creator of the segment, and a comment field describing the segment.

A video sequence is a (named) abstraction over video segments. The video sequence manager maps a user-oriented name onto an ordered list of video segment names. Video sequences can therefore be used to compose named "programs" of video segments. Information held in a video sequence descriptor includes the names of its component video segments, the creator of the sequence, and a comment field describing the sequence.

The VOM Service was implemented on our testbed as a layer of software residing between Muse and Galatea. This layer provides a procedural interface which, though higher-level than Galatea's interface, is syntactically very similar. Testbed-specific code (*e.g.*, the client stubs invoked to trigger remote operations on the VOM Service) is encapsulated behind this interface. Consequently, Muse programmers can use the new interface without knowing the testbed (or distribution) is involved. As with the E911 application, the VOM Service takes advantage of replication of managers to provide a highly-available name service; the use of replication is completely transparent to Muse application programmers.

8 A video frame can be thought of as a still image belonging to a sequence which when played in succession achieves the effect of motion video.

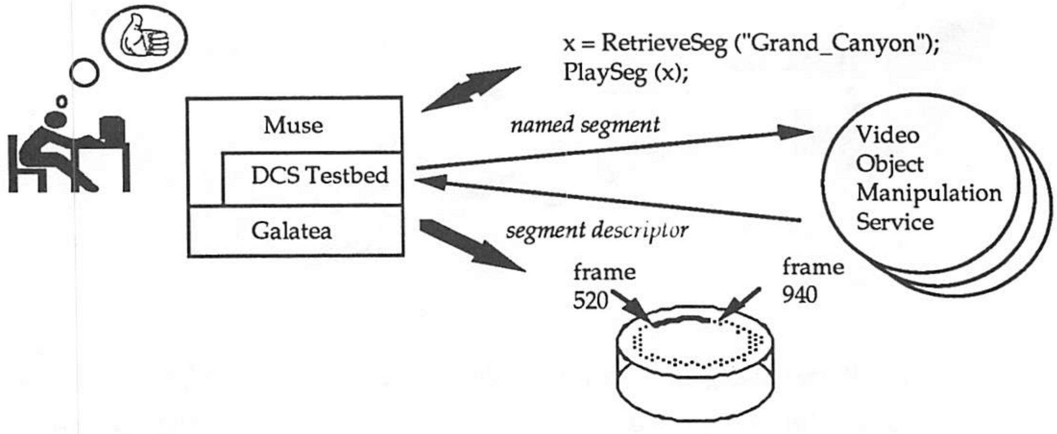


Figure 4: Muse Programming Following Introduction of Video Object Manipulation Service

Figure 4 illustrates a logical perspective of this new environment from the viewpoint of the Muse applications programmer. The programmer can now refer to video objects by name. From the name, the VOM Service can retrieve the low-level descriptor information; this is then used in the direct invocation of primitives provided by the low-level Galatea interface. It should be noted that Muse applications can bypass the testbed code, if desired, by using the standard Galatea interface directly. Through the use of the VOM Service it is therefore possible to overcome the drawbacks listed earlier. For example, by referring to video objects using more user-oriented names, Muse applications can be written to be independent of the physical positions of video data on disk. Furthermore, reuse of existing video objects is now encouraged.

Two interfaces to the VOM Service have been implemented.

The first interface is an interpreter providing full access to the underlying VOM Service functionality. Using the interpreter, named video object descriptors can be created, destroyed, and manipulated in various ways (*e.g.*, to display the video associated with a named descriptor, or to retrieve descriptors of objects satisfying simple SQL-like queries).

The second (VideoFile) interface, built on Muse, is considerably higher-level than the interpreter interface mentioned above. Using the VideoFile, "video programmers" have the ability to create, play, browse, and edit named video objects.[9] The VideoFile interface, presented in Figure 5, is very similar in nature to a standard VCR interface. A volume can be selected by clicking on any available names under the `Volume` sub-window. Video segments available on the selected volume are then listed under the (scrollable) `Segment` sub-window. Once selected, the start and end frames of a video segment are displayed in the two sub-windows towards the lower left hand corner of the interface. The segment can then be played, rewound, fast forwarded, etc., using the VCR-like buttons. VideoFile can also be used to populate the service with new video segments. For example, to create a new segment a user

9 The VideoFile interface presently abstracts over functionality supported by the VOM Service, and otherwise accessible via the interpreter interface—this includes the ability to manipulate video sequences or to perform associative retrieval of video objects.

segment a user would browse through the disk until the start frame of a segment of interest is found (the frame at the current position of the disk can be displayed at any time by clicking over either of the high-resolution image windows). The user would then click on the `Set In Point` button. In a similar way the user would scan further to find the tail end of the desired segment, finally clicking on the `Set Out Point` button. To register the range of frames bounded as a new segment, the user would finally click on the `Register Segment` button. VideoFile obtains the name of the segment and a comment string from the user through the use of pop-up windows. Once registered, the newly created segment name will appear under the list of available segments. Its descriptor is stored persistently with the VOM service.

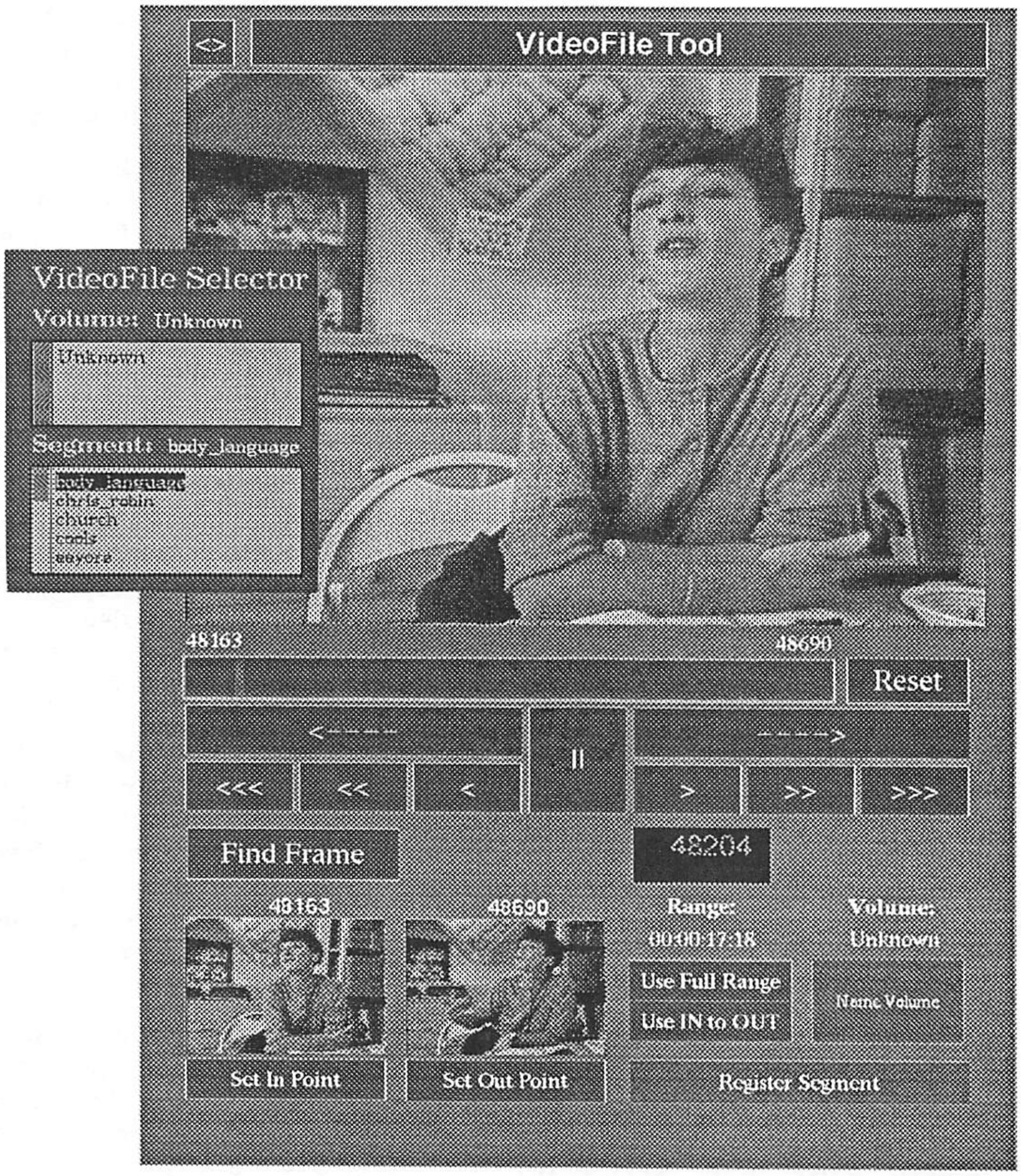


Figure 5: The VideoFile Interface to the Video Object Manipulation Service

It should be noted that practically every interaction through the VideoFile interface involves exchanges between some combination of Muse, the testbed, and Galatea, although this is highly transparent to Muse programmers.

5. Experiences with the Testbed

The bulk of this paper has centered around our initial experiences with attaining interoperability using the testbed. In this section, we summarize and discuss these experiences.

As mentioned in Section 4, our chief goals were: to experiment with the re-engineering of existing applications to address issues of distribution and heterogeneity; to study how such applications might be enhanced to take advantage of features such as fault tolerance which distributed environments have to offer; and to investigate novel applications in the telecommunications environment. The applications described in Section 4 have demonstrated the attainability of these goals.

In our use of the testbed, we have found that the guest layer approach is a very effective tool for addressing heterogeneity. Our environment exhibits heterogeneity at the hardware level (Sun and VAX nodes), operating system level (VMS and various flavors of UNIX), and application level (Oracle and Sybase DBMSs, Muse, and Galatea). The guest layer insulates applications from differences across these levels, *e.g.*, by providing a uniform virtual operating system interface, an external hardware-independent data representation, and a mechanism for inter-application cooperation. The use of the guest layer approach also allowed us to implement the applications described here without modifying existing software, thus satisfying a major real-world constraint on our environment.

Re-engineering of the E911 example resulted in an elegant decomposition of the original system into a confederation of cooperating elements. The enhanced application supports dynamic manipulation of subscriber information separately from the switch. By encapsulating the Galatea video server in the Video Object Manipulation Service, we provided an enhanced environment (via new application code) and an enhanced interface to this environment (via the Muse construction kit). The use of replicated object managers enhanced the availability and performance of both applications. In the case of the VOM service, for example, the distribution transparency achieved as a result of manager replication was key to hiding the involvement of the testbed from video users. Together with the facilities provided by the testbed, the use of encapsulation techniques made these enhancements relatively easy to achieve; also, the ability to wrap a veneer of "manager" software around existing code, such as a commercial database or video controller interface software, made it simple to attain a level of interoperability between existing and new software elements.

The ease with which the two applications were constructed from existing resources is characterized by the small amount of time and new application code required for their production. The E911 service required on the order of three man-days to integrate, and the VOM service about ten man-days. Correspondingly, in the case of the E911 service, on the order of 100 lines of C code were written, while the base functionality (excluding user-interface code) of the VOM service required about 500 lines. (These line counts do not include code generated automatically by Cronus tools.) In neither case was it necessary to modify any previously existing software.

The main strengths of our (Cronus-based) testbed include its adherence to the object-oriented "mind-set" encouraged by its model (and all the software engineering benefits this can bring), its low disruptive impact on existing software environments, its adaptability to heterogeneous environments, and its ability to support rapid application development. Although the guest layer incurs a performance penalty over the host system approach, performance of the testbed is quite adequate; its overhead is characterized by a time on the order of 20 msec for a null RPC.[10] This performance, albeit an order of magnitude slower than the "the world's fastest distributed operating system" [van Renesse88], was found to be acceptable for many of the distributed processing activities we are embracing, though inadequate for dealing with hard real-time problems found, for example, in some switching applications. In our experience, in a massively-heterogenous industrial environment it is crucial not to over-emphasize the importance of performance at the expense of other goals, such as attaining a functioning system at reasonable cost.

Experience accumulated from use of the testbed has been positive and, as such, we firmly believe the guest system approach provides a useful and expedient means for addressing many of the complex issues of interest in the telecommunications environment. The main lesson learned in this work is that current distributed systems technology has matured to the point that, with the proper tools, our goals are attainable—and with reasonable effort.

6. Conclusions

In this paper, we have conjectured that modern distributed systems and software engineering techniques hold the key to addressing the large scale heterogeneity, software maintenance, and evolution problems faced by large organizations such as GTE. As stated earlier, the attainment of interoperability has been a key goal of our project. The encapsulation property of object technology has proved invaluable in achieving this goal with minimal modification of existing components. We also argued that the guest system approach was preferable to the host system approach for our purposes; our experiences, as detailed in Section 5, support this argument. We have found that it is possible to accomplish relatively difficult goals in an extremely challenging environment using current distributed systems technology.

We have commenced work on a pilot project concerning a larger-scale, real-world application in association with a GTE business unit. This exercise will give us experience in migrating the distributed systems technology described here into a field situation, as well as providing a more stringent test of the testbed's capability of modelling the characteristics of the telecommunications environment.

In the longer term, we intend to address research issues embracing the support of very large scale, complex object systems. This work will require experimentation with a broad spectrum of mechanisms, policies, applications, and engineering trade-offs. So far we have achieved a coarse grain of interoperability; building on this, we intend to investigate finer degrees of interoperability, on the level of subsystems rather than complete systems. Other topics of interest include the applicability of per-object policies such as recovery [Wilkes87] and replication management [Wilkes88]. In parallel with these activities, we are tracking developments in OSF's

10 Between two SPARCstation IPCs over a 10 Mbps Ethernet.

Distributed Computing Environment, as well as relevant standards efforts (in particular, Open Distributed Processing and the Object Management Group).

Availability

For further information on the applications described in this paper, contact John Nicol at 617/466-4229 or Tom Wilkes at 617/466-4122, or use the postal/E-mail addresses noted at the beginning of the paper.

The Cronus 2.0 distributed computing environment is available from BBN Systems and Technologies, 10 Moulton Street, Cambridge, MA 02138. BBN can provide literature, evaluation copies, and complete support services. For further Cronus information contact James C. Berets at 617/873-2593 or jberets@bbn.com; or the Cronus Hot Line at 617/873-2111 or cronus-help@bbn.com.

Acknowledgements

We thank Russ Sasnett, Prasad Sistla, Jose Diaz, Vincent Phuah, and George Kwok for their cooperation in the work described here; also, thanks to Jim Berets of BBN for his valuable comments on an earlier draft. We particularly note the contributions made by Sam Fung in the application of distributed systems technology to the Enhanced 911 Service.

References

[Appelbaum90]
Appelbaum D A, "The Galatea Network Device Control System," MIT Media Laboratory, Cambridge MA, 1990.

[Hodges89]
Hodges M E, Sasnett R M, Ackermann M S, "A Construction Set for Multimedia Applications," *IEEE Software*, pp. 37-43, January 1989.

[Nicol86]
Nicol J R, "Operating System Design for Distributed Programming Environments," *Ph.D. Thesis*, University of Lancaster, United Kingdom, October 1986.

[Nicol88]
Nicol J R, Blair G S, Walpole J, "A Model to Support Consistency and Availability in Distributed Systems Architectures," *Proc. IEEE Conf. on Future Trends of Distributed Computing Systems in the 1990s*, pp. 417-425, Hong Kong, September 1988.

[Notkin88]
Notkin D, Black A P, Lazowska E D, Levy H M, Sanislo J, Zahorjan J, "Interconnecting Heterogeneous Computer Systems," CACM , **31** (3), pp. 258-273, March 1988.

[Robrock91]
Robrock R B, "The Intelligent Network—Changing the Face of Telecommunications," *Proceedings of the IEEE*, **79**(1), pp. 7-20, January 1991.

[Satyanarayanan90]
Satyanarayanan M, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, **23**(5), pp. 9-21, May 1990.

[Schantz86]
Schantz E S, Thomas R H, Bono G, "The Architecture of the Cronus Distributed Operating System," *Proc. 6th IEEE Int. Conf. on Distributed Computing Systems*, pp. 486-493, May 1986.

[van Renesse88]
van Renesse R, Mullender S J, Tanenbaum A S, "Performance of the World's Fastest Distributed Operating System," *Operating Systems Review*, **22**(4), October 1988.

[Vinter89]
Vinter S T, "Integrated Distributed Computing Using Heterogeneous Systems," *SIGNAL*, pp. 157-162, June 1989.

[Wilkes87]
Wilkes C T, "Programming Methodologies for Resilience and Availability," *Ph.D. Thesis*, Georgia Institute of Technology, August 1987.

[Wilkes88]
Wilkes C T, LeBlanc R J, "Distributed Locking: A Mechanism for Constructing Highly Available Objects," *Proc. 7th IEEE Symp. on Reliable Distributed Systems*, pp. 194-203, Columbus, OH, October 1988.

Efficient Order-Dependent Communication in a Distributed Virtual Memory Environment

Douglas Comer
Computer Science Department
Purdue University
West Lafayette, IN 47907
comer@cs.purdue.edu

James Griffioen
Computer Science Department
University of Kentucky
Lexington, KY 40506
griff@ms.uky.edu

Abstract

The remote memory model defines a distributed system in which dedicated, large memory machines provide high-speed backing storage to virtual memory systems executing on a set of client machines. This paper describes the design of a high-performance, reliable, communication protocol used to transfer data between clients and memory servers. To improve efficiency without affecting paging semantics, the protocol uses a partial ordering on the set of messages to allow early delivery of messages that arrive out-of-order. In an attempt to maximize throughput, the protocol supports data streaming, allowing the virtual memory system to issue multiple, concurrent, paging requests. Finally, the paper presents experimental results taken from a prototype implementation that demonstrate a substantial improvement in client performance as a result of data streaming.

1 Introduction

Virtual memory operating systems afford programmers the luxury of developing applications that assume an arbitrarily large memory space. The operating system maps certain regions of the virtual address space to physical memory and maps the remaining regions to the backing store (typically a magnetic disk).

Conventional distributed systems execute a virtual memory operating system on each node in the system. Each virtual memory system independently manages its local, private memory. No mechanism exists for the virtual memory system on one machine to use the unused memory on another machine. When the applications exceed the capacity of the local memory, the virtual memory system moves application text and data to the machine's private backing store instead of using unused memory elsewhere in the system.

Current trends in memory technology create the possibility of machines with large memories. We envision a distributed system with enough physical memory to back much, if not all, of the virtual memory being used by applications throughout a distributed system. In such a system, processes executing on the individual nodes of the system collectively share the memory resources of the the system. One such model is the *remote memory model* in which dedicated large memory machines provide a large shared memory resource to the nodes comprising the distributed system.

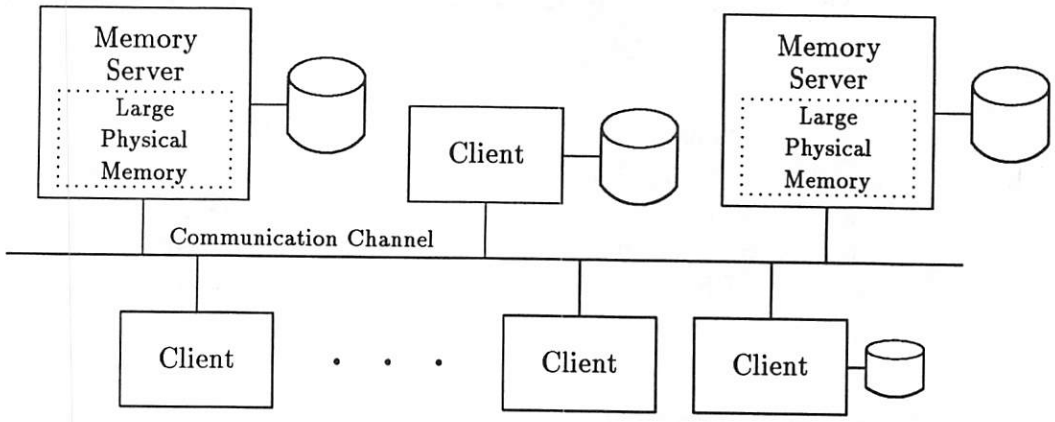


Figure 1: An example remote memory model architecture.

This paper describes the design of a remote memory communication protocol used to efficiently transfer data between machines in the remote memory model. Client machines use the remote memory communication protocol to store and retrieve data to and from remote memory. To improve efficiency but still guarantee in-order delivery of order-dependent requests, the protocol uses a partial ordering on the set of messages to allow early delivery of messages that arrive out-of-order. In an attempt to maximize throughput, the protocol supports data streaming, allowing the virtual memory system to issue multiple, concurrent, paging requests.

This paper begins with a brief overview of the remote memory model [CG91, Gri91] followed by assumptions about model and the communication channel interconnecting the machines in the model. The paper then describes our design goals and the methods used to achieve those goals. The last section of the paper presents experimental results obtained from a prototype implementation.

2 The Remote Memory Model

The *remote memory model* describes a new architecture for designing distributed systems. The model proposes a new virtual memory architecture that enhances both the functionality and the performance of distributed systems.

Figure 1 illustrates an example remote memory model architecture. The system consists of multiple *client* machines, one or more *memory server* machines, various other servers (e.g., time servers, name servers, or file servers), and a communication channel interconnecting all the machines. In the remote memory model, memory server machines provide additional memory storage space for the virtual memory systems executing on the client machines. Instead of assigning the memory resources to individual machines where the memory is installed as private memory, the remote memory model provides a large memory resource which all the clients share. In the remote memory model, client machines access the memory server across a high-speed communication channel to obtain additional storage space.

Each client machine has a local memory capable of satisfying the client's normal processing demands. However, for jobs requiring large amounts of memory, clients use the large memory of the memory server for additional storage space. When the memory requirements of an application exceed the capacity of the local memory, the client's virtual memory system stores some of the application's data on the memory server. When the application attempts to access data not cached in the local memory, the virtual memory system intercepts the access, transfers the desired data from the memory server to the client's local memory, and resumes execution. Each client's local memory functions as a high-speed cache of the large, shared, memory resource on the memory server.

Large memory machines called *memory servers* provide shared, high-speed remote memory storage to the virtual memory systems executing on the client machines. Memory servers are dedicated machines whose sole purpose is to provide high-performance remote data storage. Each memory server machine has a large amount of storage space consisting of physical memory and one or more mass storage devices (e.g., disk drives).

The remote memory model approaches virtual memory differently than conventional distributed systems by allowing clients to share a large, globally accessible, memory resource. Memory servers act as monitors for the memory resource, allocating memory to clients based on their needs. The remote memory model also provides the client virtual memory systems with the opportunity to interact via shared data. The above functionality differences together with high performance make the remote memory model an attractive alternative to conventional distributed systems.

3 Assumptions

In the remote memory model, clients use the communication channel to access the additional memory storage space provided by the memory server. Because the efficiency of the communication protocol significantly impacts the performance of the entire system, we designed a special purpose communication protocol called the *Remote Memory Communication Protocol* (RMCP). Clients use the protocol to transfer data to the memory server when they need additional memory space, and the memory server uses the protocol to transfer data back to the client when the client requests the data.

Before examining the design of the remote memory communication protocol, we must first list our assumptions regarding the communication channel and the machines it connects:

Connectivity: The communication channel provides message delivery between a memory server and each client machine. If necessary, the communication channel handles routing (e.g., across gateways) to provide a transmission path between the server and a client.

Datagram Service: The communication channel provides datagram (packet) oriented service.

Unreliable Delivery: The communication channel unreliably delivers packets. The communication channel may drop, corrupt, or delay packets. However, the rate at which these errors occur is relatively low.

High-Speed: The communication channel offers a relatively high bandwidth and low delay (e.g., a 10 Mbps Ethernet, 100 Mbps FDDI, or faster).

Heterogeneous Clients: The distributed system consists of heterogeneous client architectures and operating systems.

Concurrent Server Access: The memory server serves multiple client machines simultaneously, providing concurrent access to the remote memory resource.

This small set of assumptions serves as the basis for the design of the Remote Memory Communication Protocol. The following section outlines our goals for the Remote Memory Communication Protocol and the effect our assumptions have on the design of the protocol.

4 Design Goals

RMCP serves as the client virtual memory system's interface to the memory server machines. In some respects the RMCP protocol functions much like an RPC style protocol [BN84, Lyo84, Wel86, Che88], storing and retrieving data to and from the memory servers. In addition to the basic goal of send and retrieving data, we identified the following three design goals for the RMCP protocol:

Reliability: To insure correct operation of the client virtual memory system, the communication protocol must reliably transfer data to and from the memory server. In addition, the protocol must guarantee that the server will perform the store and fetch requests in the order the client virtual memory system issues them (i.e., client requests are order-dependent).

Architecture Independence: The recent proliferation of network architectures and computer architectures, combined with a desire to execute the communication protocol across a wide variety of hardware platforms, mandates a communication protocol design that limits the number of modifications required to port the protocol to new architectures.

Efficiency: To improve the virtual memory system's performance, the communication protocol should:

- minimize the number of messages sent between a client and a server when executing a store or fetch request,
- minimize the number of packets used to transmit a message across the communication channel,
- minimize the per-message processing overhead on the client and the server (e.g., processing the protocol headers),
- take advantage of the asynchronous nature of the underlying communication channel whenever possible by issuing multiple, concurrent, requests.

Unfortunately, the reliability objective often results in additional messages and added overhead, reducing the efficiency of the protocol. The tradeoff between reliability and efficiency makes it difficult to find the correct balance that will optimize overall system performance. However, because we assume the underlying communication channel has a relatively low error rate, we cater toward efficiency and optimize the protocol's performance for the most common case: the case when no errors occur.

5 Conceptual View Of The RMCP Protocol

The *Remote Memory Communication Protocol* is a special purpose protocol designed to meet the reliability, efficiency, and architecture independence requirements described in the previous section. The protocol provides reliable delivery, data streaming, arbitrarily large messages that permit arbitrary page size transfers, independence from the underlying network architecture, and low overhead that results in high efficiency. All client machines use RMCP to transfer data to and from the memory servers.

The Remote Memory Communication Protocol consists of two layers: the *Xinu Paging Protocol* (XPP) Layer and the *Negative Acknowledgement Fragmentation Protocol* (NAFP) layer. Figure 2 illustrates the layering and the flow of data through the layers.

Dividing the protocol into two distinct layers clearly separates the tasks performed by the protocol into high-level *paging* operations and low-level *transport* operations. The XPP layer implements the high-level abstract paging operations used to reliably store and retrieve data to and from the memory server. The NAFP layer implements the low-level transport operations required to transfer XPP messages across the communication channel. NAFP hides the characteristics of the communication channel and the details of message delivery from the XPP layer, allowing the XPP layer to remain independent from the underlying network architecture.

Both the XPP layer and the NAFP layer adhere to the network protocol layering principle [Com88]. The NAFP layer provides many of the standard transport level services such as fragmentation, message queueing, and an architecture-independent interface to the underlying communication channel. In addition, NAFP uses negative acknowledgements to provide early error detection/correction. For the purposes of this paper, we will omit the details concerning the NAFP layer[1], and instead focus on the design of the XPP component of the protocol.

6 The XPP Protocol

The Xinu Paging Protocol provides the virtual memory system with the ability to issue high-level abstract paging operations. XPP hides the details of backing store I/O from the virtual memory system by reliably transmitting paging messages, invoking the desired paging operation on the memory server, and returning the results. The XPP protocol packages up the requested operation into an XPP message and invokes NAFP to deliver the message across the communication channel to the memory server. The XPP layer on the

[1]A description of the NAFP protocol can be found in [CG90].

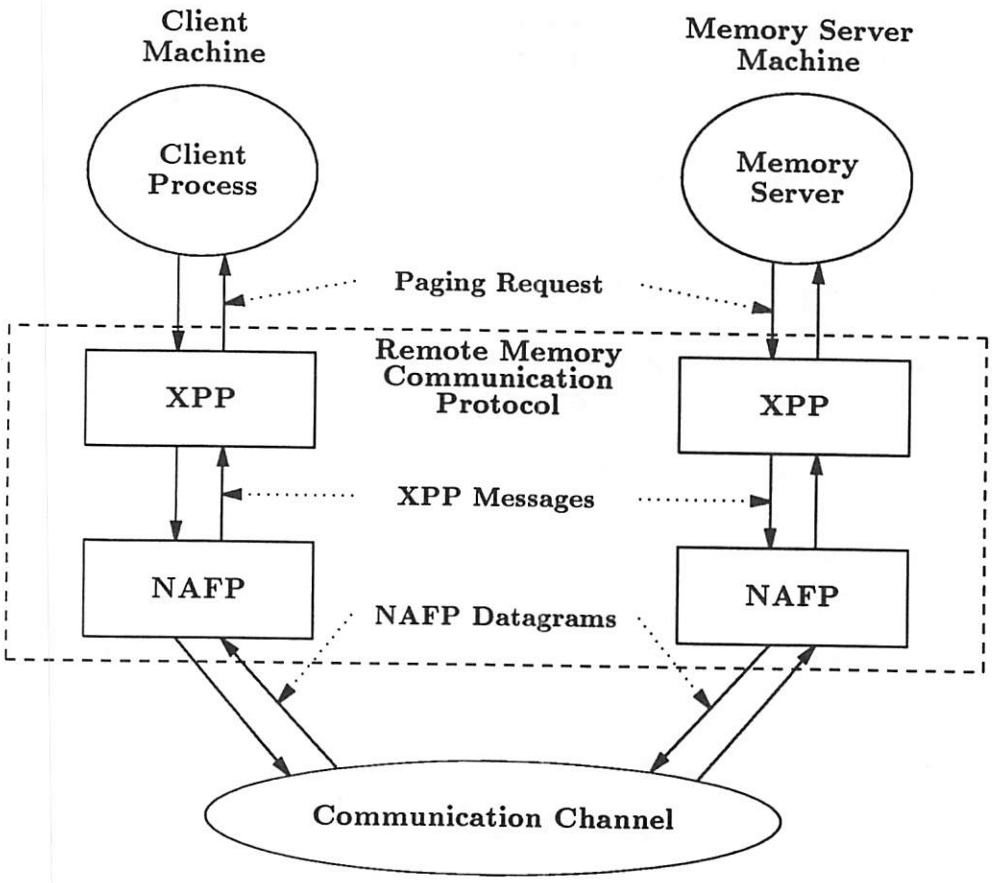


Figure 2: The two layers of the remote memory communication protocol. A client initiates an XPP request which travels down through the layers, across the communication channel, and back up through the layers to the memory server. The memory server processes the request and sends an XPP reply back through the layers and across the communication channel to the waiting client process.

server unpacks the XPP message to obtain the original operation and delivers the requested paging operation to the memory server. When the server completes the operation, it sends an XPP reply message to the client to inform the virtual memory system that the operation has finished. If the NAFP layer fails to deliver an XPP request message, the XPP layer will retransmit the message until it succeeds. Because the XPP protocol provides reliability, the virtual memory system only needs to issue a store page or fetch page operation once and wait for the results.

6.1 XPP Message Types

The XPP protocol supports four basic message types which the virtual memory system uses to access the memory server. Each message type corresponds to one of the memory server's abstract paging operations.[2] The four abstract paging operations provide the functionality needed to support the virtual memory system. These four basic message types are:

- page store requests
- page fetch requests
- process create requests
- process terminate requests.

The client's virtual memory system uses page store request messages and page fetch request messages to send and receive data to and from the server. When a client creates or terminates a process, the virtual memory system informs the memory server using a process create request or a process terminate request. The server uses this information to provide efficient storage and to detect erroneous paging requests.

6.2 Reliability

The XPP protocol uses timeouts, retransmissions, and positive acknowledgements to satisfy the end-to-end reliability requirement imposed by the client virtual memory systems. Positive acknowledgements, in XPP, function differently than positive acknowledgements in protocols that guarantee reliable delivery. Positive acknowledgements in protocols that guarantee reliable delivery indicate that the destination received the message [Com88]. In XPP, the reply to a request message acts as a positive acknowledgement. In addition to acknowledging receipt of the request message, a reply message indicates that the request message was processed by returning the status of the requested operation. XPP starts a timer when it sends a request message and retransmits the request message if the timer expires before the reply message arrives. XPP saves a copy of the message and does not delete the request until it receives a successful reply. Each time XPP resends the request message, it resets the timer and waits for a reply.

[2] A detailed description of the memory server and the operations supported can be found in [Gri91].

6.3 Message Sequencing

Virtual memory systems, including VM Xinu's virtual memory system, expect paging requests to be processed in the same order that the requests were issued [PS85, LKKQ89]. Consequently, XPP must guarantee to deliver paging messages to the memory server in-order. If the server does not process the messages in the order they were issued, the server may return incorrect data in response to a fetch request.

XPP uses sequence numbers to guarantee that request messages are delivered in the correct order. A sequence number serves two purposes: it uniquely identifies a message, and it imposes an ordering on the list of messages. The client assigns a sequence number to each XPP message it sends to the server. The client and the server agree on an initial sequence number and increment the sequence number for each new request message sent. Each XPP client/server pair has its own sequence number.

Because XPP assigns sequence numbers in increasing order, the sequence numbers impose an ordering on the list of messages. XPP uses sequence numbers to deliver messages to the server in an order that insures correct results. However, the virtual memory system does not need to impose such a strict ordering to achieve correct results. In practice, the virtual memory system only needs to impose a partial ordering on the list of messages to achieve correctness. That is, the case seldom occurs where the server must process message i before processing message $i + 1$. More often the case arises where the server can process message $i + 1$ before message i and still achieve the same results. For example, assume a client issues a store request for physical page i belonging to process A followed by a fetch request for physical page j belonging to process B. Because the two messages do not depend on each other in any way, XPP may deliver the messages out of order without affecting the result. Consequently, XPP will, under certain circumstances, pass out-of-order messages on to the server in an attempt to improve efficiency.

Each XPP message includes a *preceding message number* which XPP uses to impose a partial ordering on the list of messages. We say that message j depends on message i if the server must process message i before processing message j. We define the relation $\preceq$ on the set of messages as

$$m_i \preceq m_j \text{ if } m_j \text{ depends on } m_i, \text{ or if } i = j \;.$$

The relation $\preceq$ defines a partial ordering on the list of messages $m_1, m_2, ..., m_n$. XPP remembers the last sequence number the server processed. When the XPP layer on the server receives a new request, it checks the sequence number against the last sequence number to see if the server missed any request messages. If XPP receives a request message out of order, it uses the partial ordering to determine whether the server should process the out of order message anyway or wait until the missing request arrives.

Because clients issue messages with increasing sequence numbers, m_i does not depend on m_j for $i < j$. Consequently, for a given message m_j, XPP only needs to verify that

$$m_i \preceq m_j \text{ is false for all } i,\ l < i < j \;.$$

where l is the last sequence number processed. XPP uses the preceding message number to perform the verification in a single operation. We define the preceding message number as

$$p = \ \max i, \text{ such that } i < curmsg \text{ and } m_i \preceq m_{curmsg} \;.$$

That is, the preceding message number specifies the sequence number of the most recent preceding message that must be processed before the current message can be processed. Because the client virtual memory system knows the type and content of each XPP request message, it computes and sends a preceding message number with each XPP message. XPP compares the preceding message number (p) with the sequence number of the last message the server processed (l). If $p \leq l$, XPP gives the message to the server for processing rather than allowing the server to sit idle waiting for the missing message(s).

The concept of a preceding message number is a powerful idea that can be applied to other protocols as well; in particular, protocols where some, but not all, messages must be delivered in-order. That is, preceding message numbers can be use improve the performance of any protocol for which there exists a partial ordering on the list of messages. For example, imagine a network windowing system protocol (e.g., a protocol similar to the X or NeWS protocols[Nye90, Sun]) in which a partial ordering exists on the list of screen manipulation messages. Preceding message numbers would allow the system to process independent screen manipulation messages out of order (e.g., two independent draw_line messages), but yet insure that order-dependent operations are performed in the correct order (e.g., a blank_screen message followed by a draw_line message).

6.4 Data Streaming

XPP uses data streaming to increase concurrency and total throughput. Instead of sending one message at a time, XPP allows the virtual memory system to issue several XPP messages concurrently. To use as much of the network bandwidth as possible, XPP sends a stream of messages to the server.

XPP maintains a *pending list* of request messages that the server has not yet processed or acknowledged with an XPP reply message. To avoid overrunning the server with request messages, XPP limits the length of the pending list, only allowing a finite number of outstanding requests at any given time. XPP sends multiple request messages back-to-back until it fills the pending list. The client then waits for the server to process the messages and reply. When the client receives an XPP reply message corresponding to a request message on the pending list, the client removes the request message from the pending list and sends the next waiting request message.

Figure 3 compares data streaming to synchronous transmission and shows the effect of data streaming when using a pending list of length three. In the synchronous case, the paging system spends much of its time idle, waiting for messages. Data streaming, however, keeps the client and the server busy and increases throughput.

6.5 A Prototype Implementation

We designed and implemented a prototype distributed system based on the remote memory model. Each client machine executes the VM Xinu operating system and uses the RMCP protocol to communicate with the memory server machine. The client machines consist of Sun Microsystems Sun 3/50's, Digital Equipment Corporation MicroVAX I's, II's, and DECstation 3100's. A 10 Mbps Ethernet interconnects the client machines and the memory server. The prototype memory server executes on a SPARCstation 1.

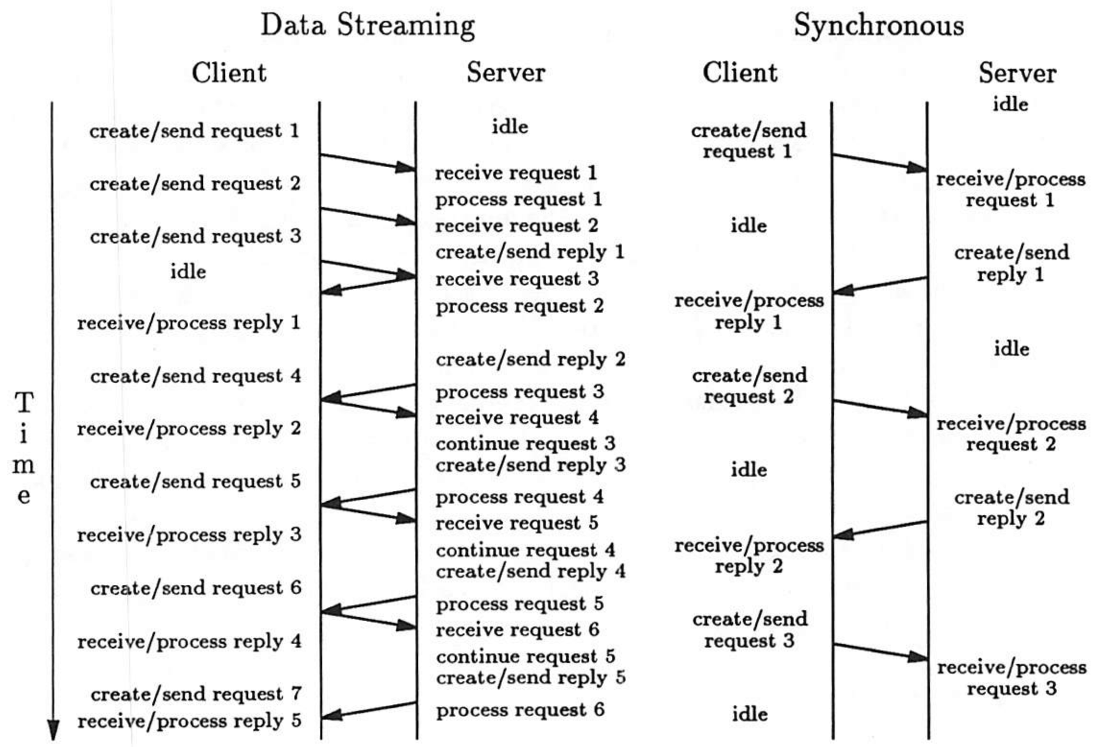


Figure 3: Data streaming vs. synchronous sends. Data streaming with a pending list of length 3 keeps the system busy, while synchronous sends result in a substantial amount of idle time. In each case, time moves down the page and shows the messages transferred.

6.6 Experimental Results

The Sun, MicroVAX, and DECstation client machines simultaneously access the remote memory server for backing storage, demonstrating support for heterogeneous clients. To allow operation over a wide variety of networks, our communication protocol uses UDP [Pos80] as the underlying datagram transport protocol. Our initial measurements indicate that our implementation choice of UDP as the underlying communication channel accounts for at least half of the total processing time [CG91, Gri91]. Despite the added overhead of UDP, our initial timing results show that storing or retrieving an 8K byte segment between a Sun 3/50 client and a memory server executing on a SPARCstation 1 requires 20 ms (average). In comparison, our measurements show that the average time for a Sun 3/50 to store or fetch an 8K byte page to or from a local disk is 22 ms.

6.6.1 Data Streaming

Our experience indicates that most virtual memory systems exhibit bursty backing store I/O behavior. Under normal processing demands the virtual memory system rarely pages to the backing store. However, when a user executes a program requiring a large amount of memory the virtual memory system suddenly transfers large amounts of data to the backing store. Also, whenever the user resumes a process that has been swapped out, the virtual memory system suddenly transfers a large amount of data between memory and the backing store. The long periods of infrequent paging interspersed with sudden flurries of paging activity emphasize the need for data streaming.

The length of the pending list controls the amount of data streaming. Although data streaming allows the virtual memory system to use more of the network bandwidth, choosing an optimal pending list length is difficult and depends on several factors. One might think that increasing the pending list length would always improve the client's performance because the virtual memory system can use more of the network bandwidth. However, Figure 4 indicates that our intuition is not necessarily correct. Figure 4 shows the total runtime (in seconds) for a test application that repeatedly traverses its large virtual space in a sequential manner. The total runtime decreases when we change the pending list length from 1 to 2; however, the total runtime increases when we go to a pending list length of 3. To understand the anomaly in the graph, one must examine the application (in this case the sequential access test) and the behavior of the page replacement algorithm.

The sequential test program incurs page faults at a steady and rapid rate, performing almost no computation between faults. Consequently, the number of free pages drops to the low water mark and the replacement algorithm executes continuously, trying to reclaim memory. When the pending list length is 1, the virtual memory system synchronously issues a fetch request in response to a page fault. As soon as the fetch request completes, the virtual memory system issues a store request to replenish the free list. While waiting for the store request to complete, the test program faults again, and the cycle continues (see Table 1). With a pending list of length 2, the virtual memory system issues a fetch request and a store request back-to-back. When the client receives the fetch reply it immediately sends a second store request. Because the virtual memory system can issue a store request while waiting for the fetch reply, the application's total runtime decreases. When the pending list

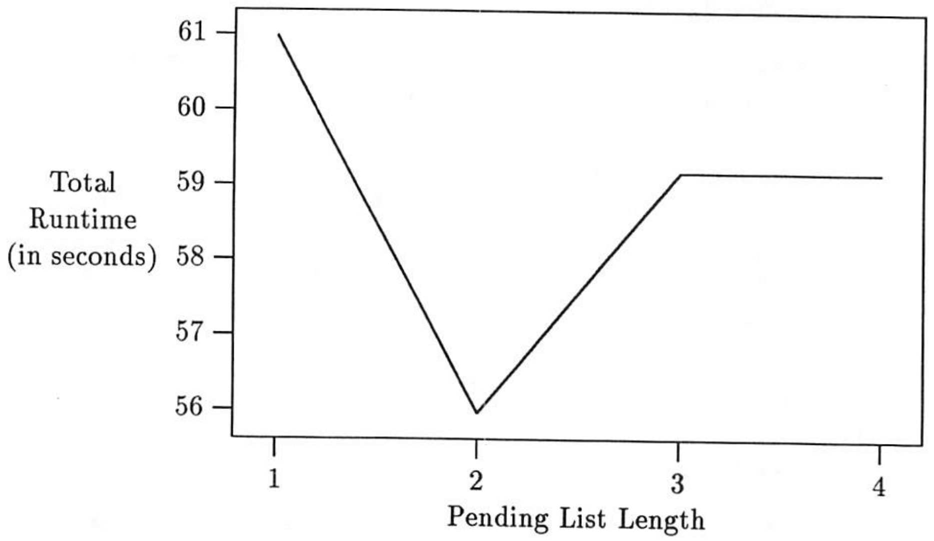


Figure 4: The total run time of the sequential access test program as a function of the pending list length.

length is 3, the virtual memory system asynchronously issues a fetch request followed by two store requests. While processing the second store request, the client receives the fetch reply and issues a third store request. Because the page replacement policy aggressively frees pages, the virtual memory system issues 3 store requests between each fetch request as opposed to the 2 store requests when the pending list length is 2. Although the total amount of data being transmitted between the client and the server increase by going to a pending list of length 3, the fetch requests become more spread out, thus increasing the total runtime of the application. The type of application, the page replacement policy, the scheduling policy, and the server response time all affect the choice of an optimal pending list length. However, the figure clearly shows that even a small pending list of length 2 substantially improves the client's performance. In this case, data streaming results in an 8.3% improvement in total execution time over synchronous delivery.

Pending List Length	Request Pattern
1	fsfsfsfsfsfs
2	fssfssfssfss
3	fsssfsssfsss
4	fsssfsssfsss

Table 1: Sequence of requests observed at the memory server for various pending list lengths. The letter f denotes a fetch request and the letter s denotes a store request.

7 Conclusions

Our initial test results inidicate that the use of an efficient communication protocol executing on existing network hardware can provide performance competitive with that of conventional backing storage. Through the use of preceding message numbers and data streaming, the RMCP protocol is able to provide highly-efficient, reliable, ordered communication between client machines and memory servers. Although, our test results show that choosing an optimal pending list length can be difficult, the results clearly show that data streaming (i.e., a pending list length of 2 or more) can substantially improve the performance of a client's virtual memory system. In addition, the remote memory model capitalizes on recent, and continuing, advances in processor speeds, network bandwidth, and memory sizes and appears promising as a design model for distributed systems in the future.

8 Availability

VM Xinu source code for the Sun 3 architecture is currently available. For more information, contact griff@ms.uky.edu or send a request to xinu-info-request@cs.purdue.edu.

9 Acknowledgments

We would like to thank Gouben Li for his help with the implementation of the prototype system.

References

[BN84] A.D. Birrel and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CG90] Douglas Comer and James Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135. USENIX Association, June 1990.

[CG91] Douglas Comer and James Griffioen. Virtual Memory Xinu. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems.* USENIX Association, March 1991. Also released as Technical Report No. CSD-TR-1028, Department of Computer Science, Purdue University.

[Che88] David Cheriton. VMTP: Versatile Message Transaction Protocol. ARPANET Working Group Requests For Comments, Febuary 1988. RFC 1045.

[Com88] Douglas Comer. *Internetworking With TCP/IP: Principles, Protocols, and Architecture.* Prentice Hall, Inc., 1988.

[Gri91] James Griffioen. *Remote Memory Backing Storage For Distributed Virtual Memory Operating Systems.* PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, August 1991.

[LKKQ89] Samuel J. Leffler, Marshal K. Mc Kusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison Wesley, 1989.

[Lyo84] Bob Lyon. Sun Remote Procedure Call Specification. Technical report, Sun Microsystems, Inc., 1984.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[Nye90] Adrian Nye. *X Protocol Reference Manual.* O'Reilly and Associates, Inc., 1990.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[Pos80] J. Postel. User Datagram Protocol, August 1980. RFC 768.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, 1985.

[Sun] Sun Microsystems, Inc. *The NeWS Programmer's Guide.*

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distribute Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[Wel86] Brent B. Welch. The Sprite Remote Procedure Call System. Technical Report UCB/CSD 86/302n, University of California Berkeley, June 1986.

[YTR87] Michael Young, Avadis Tevanian, and Richard Rashid. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the ACM Symposium on Operating System Principles.* ACM, November 1987.

Implementing DVSM on the TOPSY multicomputer

T.Stiemerling and T.Wilkinson
Computer Science Department, City University,
Northampton Square, London EC1V 0HB, UK.
A.Saulsbury*
Department of Computing, Imperial College,
180 Queens Gate, London SW7 2BZ,UK.

Abstract

We describe the implementation of distributed virtual shared memory (DVSM) on the TOPSY multicomputer. The TOPSY machine is a distributed memory multiprocessor based on MC68030 nodes connected by a custom circuit-switched mesh interconnection network, and runs the MESHIX operating system which is System V Unix compatible. The DVSM allows distributed processes to share a paged virtual memory region, whose coherence is maintained by user-level servers using a dynamic distributed manager algorithm. The DVSM implementation is described at the user, server and kernel level, and some basic performance measurements are presented. How the DVSM is integrated with the MESHIX paging mechanism, and a spin-lock synchronisation mechanism is described. An overview of the relevant parts of the MESHIX operating systems is also given.

1 Introduction

In this paper we describe work in progress on the design and implementation of distributed virtual shared memory (DVSM) on the TOPSY multicomputer [1]. The operating system for TOPSY is MESHIX [2], which is compatible with System V Unix. The DVSM allows distributed Unix processes to share paged virtual memory regions transparently across the machine. Coherence of the DVSM region is maintained by user-level server processes based on the dynamic distributed manager algorithm of Li [3].

The motivation for investigating distributed virtual shared memory (DVSM) has been to enable systems which do not contain physically shared memory, such as a network of workstations or a distributed memory multicomputer, to be programmed using a shared memory paradigm rather than explicit message-passing. Research into DVSM is popular, over 15 different systems have been reported in the literature, and the issues are generally well known. We therefore only give a cursory review of these (for a more in-depth review see [4, 5]).

The main differences in DVSM systems relate to one of the issues of: integration, coherence, synchronisation and virtual memory. The DVSM can be integrated at a number of levels, typically hardware, kernel or server level (although there may some overlap between all these levels). Hardware level systems generally provide higher performance and finer-grain sharing of the DVSM, for example at the word or cache-line. The majority of implementations are at the kernel or user level, and build on existing memory management primitives. Various semantics for coherence, such as strict, weak, or release can be implemented. Strongly coherent systems do not require any alterations to existing parallel applications. How the coherence is maintained also varies depending on the granularity of the DVSM and the communications network used, for example a hardware system may use word update,

*Current address: Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden.

and a user level system page invalidation. Some integration between the DVSM and synchronisation mechanisms is necessary. Simple schemes such as using spin-locks directly in the DVSM may not be efficient. If large DVSM regions are to be used then virtual memory support is required. Usually this is implemented by swapping pages to and from disk. The DVSM described here is a user-level server which maintains strict coherence using page invalidation, supports synchronisation using spin-locks, and allows page swapping to disk. The implementation has similarities to systems such as IVY [3], SHIVA [6], and KOAN [7]. The use of an external-pager interface is similar to MACH [8].

The paper is arranged as follows: In the next section we describe the organisation and give an overview of the TOPSY multicomputer and the MESHIX operating system. Then we describe the DVSM implementation starting with the coherence algorithm, and go on to the user interface, the internal protocol of the DVSM server, and the additions to the MESHIX kernel. Next the integration of the DVSM into the MESHIX virtual memory paging mechanism is described, followed by a description of a spin-lock synchronisation mechanism. We then present a preliminary performance analysis of the DVSM, and compare our implementation to similar systems. Finally we conclude with our future work plans.

2 TOPSY and MESHIX

TOPSY is designed to be a low-cost extensible general purpose computer which supports standard Unix software. Below we give an overview of the TOPSY machine and the MESHIX operating system, particularly in the areas inter-process communications and memory management.

2.1 TOPSY multicomputer

The TOPSY machine is a scalable distributed memory multicomputer consisting of up to 256 processor-memory nodes connected by a mesh network [1]. Each node consists of a Motorola MC68030 processor (16 MHz clock), 8 Mbytes of memory, a network interface, and other peripheral interfaces (including SCSI, serial drivers, ethernet etc.). The interconnection network, named MESHNET [9], is a circuit-switched message-passing network with a toroidal topology, and has a maximum bandwidth of 12 Mbytes/sec per channel[1]. A 16 node machine and a number of 4 node machines have been constructed.

2.2 MESHIX operating system

MESHIX is a multicomputer operating system which is interface compatible with release 3.0 of System V Unix[2] [2]. MESHIX has been implemented using a micro-kernel approach in which the kernel functionality has been seperated into a number of privileged server processes executing concurrently on each node of the system and communicating using message-passing [10].

MESHIX provides two basic forms of message-passing communication primitive. Either fixed length messages can be transferred asynchronously between processes, or an arbitrary length memory block can be transferred synchronously between processors (*global-copy*). Messages are guaranteed to arrive in sequence between the same source and destination nodes. Messages are addressed to ports, which can be associated with particular processes.

MESHIX provides the standard Unix virtual memory model [11]. A MESHIX process has three default regions: a text region (maximum 4 Mbytes), a data region (maximum 2 Mbytes) and a stack region (maximum 8 Mbytes). When a process tries to access a virtual address that is not mapped to a core page, or violates the access permissions on a page, a page fault is generated by the MC68030 memory management unit (MMU). All such faults are handled by the kernel fault handler routine.

Two Unix kernel processes, the memory manager and swap manager, are responsible for virtual memory support. The memory manager is responsible for creating and controlling the virtual memory of user processes. The swap manager is responsible for the movement of core pages from main

[1]Currently this is reduced to about $\frac{1}{2}$Mbyte/sec effective bandwidth, partly because the network is serviced by a MC68450 DMA controller and partly due to the software overhead of message transmission.

[2]System V *and* Unix *are trademarks of Unix System Laboratories Inc.*

memory onto a swap device (a local or remote disk). The memory manager process also executes most of the Unix system calls for user processes.

To generate parallel processes on one node in MESHIX the `fork` system call is used, but to start a parallel process on another node a subsequent `exec` call must be executed. This is because process state that has to be transferred during an `exec` is minimal, and generally a `fork` will be followed by an `exec` most of the time anyway.

3 The DVSM strategy

In implementing DVSM on TOPSY we have used an *external-pager* approach similar to MACH [8]. Every node on the TOPSY machine has a user-level DVSM server process executing on it. Each server process handles the DVSM requests made by processes on its local node, and communicates with the servers on the other nodes to satisfy these requests. The DVSM servers use the well known *dynamic distributed manager* algorithm of Li [3], with invalidation to maintain per-page (strong) coherence of the DVSM.

Processes which make use of the DVSM access as part of their normal data space. No special function calls are required to interact with the DVSM, apart from initially *attaching* a process to it. The processors MMU catches page faults for the DVSM region in the same manner as for the other regions, but instead of the default MESHIX fault handler a DVSM specific fault handler is executed. This fault handler communicates with the local DVSM server, which must then service the fault.

Having a user-level server has advantages from the development point of view, since the server can be changed without having to re-compile the kernel. An external-pager interface had to be added to the MESHIX kernel first, though. The functionality currently provided by this interface was defined by the requirements of the DVSM server, and it is therefore not a general virtual memory interface such as those provided by Mach [8] or Chorus [12].

3.1 Dynamic distributed manager algorithm

In the dynamic distributed manager algorithm the server controlling a page in the DVSM can vary dynamically, and the servers are distributed around the machine. This means the work-load of maintaining coherence is also distributed around the machine, making the implementation more scalable. An overview of our implementation of the algorithm is given below.

3.1.1 Ownership

The algorithm allows multiple read-only copies of pages, but only a single writeable copy. Only one server at a time has control (or "*ownership*") of any particular DVSM page at a time, but this ownership may be transferred dynamically from server to server. Only the node which has ownership has the ability to write to a page and alter its contents. This means that ownership must be requested by any node wishing to write to any page it does not already "own". This transfer of ownership is taken care of by the "owning" node, at the request of the server on the faulting node.

When a process on a non-owning node tries to read from a DVSM page, the node either already has a (read-only) copy of the page, or has to obtain a new copy of the page. When a page is not present the local DVSM server sends a message to the owning server to supply a copy of the requested page. The owning server keeps a list of nodes which have copies of the page, called the copy set or invalidation list.

With ownership goes the responsibility of maintaining page coherence. There may be a number of read-only copies of the page which must be updated when any changes are made to the page. The owning server therefore invalidates any other copies of the page, and the other servers may then request new copies of the page.

3.1.2 Transfer of ownership

When a process on a non-owning node tries to write to a read-only copy page a fault occurs. An ownership request is then sent to the DVSM server on the node which currently owns the page. The owning server has the responsibility of invalidating any copies of the page[3], and transferring its (up-to-date) copy and ownership of the page to the requesting node. The old owner does not delete it's copy of the page, but marks it read-only. After receiving the page and ownership the new owner marks the page writeable, and the faulting process is re-scheduled to complete the write.

3.1.3 Finding the owner

When ownership is transferred the other servers are not informed of the change. Each server therefore keeps a record, not of the actual owner, but of the *probable* owner of a page. When a node transfers ownership it sets its probable owner information to be the new owner node. Requests arriving at a node for a page not owned by that node are then passed on to the probable owner. In this manner an ownership message sent to the wrong node is forwarded on until it reaches the actual owning node. The limit for the number of messages to locate the page owner in an N node system is $N-1$ [3].

3.1.4 Invalidation

If there are copies of a page besides the owning node, then the page on the owning node is marked read-only. When a process on the owning node tries to write to a page a write fault is generated. The server on the owning node then uses the invalidation list to send an *invalidation* message to each of the servers on other nodes which have a copy of the page. The invalidation causes the non-owning nodes to throw away their copies of the same page, and they then send an acknowledgement back to the owner[4]. When all the acknowledgements have been received, the page on the owning node is marked writeable and the faulted process is re-scheduled to complete the write.

4 User to server communication

Before the DVSM can be used the servers must be started up. The first instance of the server to be invoked, the *installing* server, then starts *slave* servers on all of the other nodes. Once all the servers have acknowledged that they started correctly they wait for any messages. To use the DVSM a user's program must *attach* to an existing DVSM region, or creates a new one, using the DVSM calls provided. Currently each process is limited to one attached DVSM region only, although the servers can manage a number of different regions.

4.1 DVSM user calls

Example DVSM C library function calls to make use of DVSM regions are shown in Figure 1. These functions simply send a message to the local DSVM server containing the function type and parameters.

Before a process can attach to a DVSM region it must specify which DVSM server is to be used using an initialise call, which serves to translate a symbolic server name into a message port number. This also allows the process to select one of a number of concurrently running servers, which may implement differing coherence semantics for example.

To attach to a DVSM region the process executes the attach call with the region key, access permissions, region size, and region address as parameters. The key is an integer which uniquely identifies a particular DVSM region. A DVSM region is shared between processes (assuming correct permissions), by giving the same key when attaching. When a region is first created the region size is specified in bytes and cannot be changed subsequently. The address is where the region is mapped

[3]Here we differ slightly from the mechanism of Li [3], in which the new owner invalidates the page copies after having been passed the copy set. This saves us having to transfer the copy-set as well as the page contents.

[4]If we do not require acknowledgements to be returned in response to invalidations then technically we have a weaker form of coherence, since the write may occur while there are still valid copies of the page [13].

```
int dsm_init(char * server_name)

int dsm_attach(int key, flags, size, int * address);

int dsm_detach(int key);
```

Figure 1: C function calls for initialising, attaching and detaching from a DVSM region

into the processes address field (and is currently fixed). The local DVSM server then creates a region of the given size if the region does not already exist, and maps the region into the calling processes address space.

When a process has finished using a DVSM region it executes the detach call with the region key as parameter. The region is destroyed when all attached processes across the machine have detached. If a process terminates before detaching then the usecount in the kernel region table is decremented, but the server is not notified.

4.2 Other user interfaces

The calls described above provide a basic interface to the DVSM, but only allow programs that include these calls to use the DVSM. To allow programs written for other shared memory interfaces to be ported onto TOPSY it is necessary to either re-write the relevant parts of the program, or to provide an alternative (compatible) shared memory interface[5]. An example is the Unix System V shared memory library [14], which is part of a larger IPC facility. The System V shared memory calls are similar to the one's described above but differ in a number of details, for example region creation and attachment are separated into two calls, and there is no detach call (destruction of a region is achieved using a control call). Extra functionality such as region locking is also provided.

5 Server organisation and operation

After the servers have been initialised they wait for messages to arrive from a user process, the kernel or another server. The actions of the server in response to a message are dependent on the message type and the internal server state, both described below. The server process is single-threaded, but because some operations have a long latency the server has been *pseudo* multi-threaded so it can start new operations while waiting for others to complete. For example during an invalidation, the server can service other messages while waiting for all the acknowledgements to arrive.

5.1 Server page tables

Every server maintains a number of data structures to store information about the DVSM regions. Some of this information is replicated in every server for each region. The main data structures are the region and page tables. The region table is indexed by the region key, and contains a region table entry for each region currently in use.

Each region table entry contains information about the region including the region size, the number of pages in the region, a pointer to a page table for the region, a use count for the region, the access permission flags, and a pointer to the kernel region table entry for the region on that node.

Each region has a page table which contains information about every page in the region, and is indexed by the page number. The page state defines the current state of the page. The other fields include the address of the last probable owner of the page, the number of copies of the page, the

[5] Experience so far indicates that other issues such as the process model used generate far more problems for porting than the shared memory interface.

state	mnemonic	description
0	nothing	page is not currently allocated
1	copy	page is allocated and is a read-only copy
2	owner	page is allocated and the server owns it
3	intransit	page has been requested
4	owninval	page is being invalidated by the owning node
5	transpg	page contents and ownership are being transferred
6	transown	page ownership is being transferred

Table 1: Possible page states and their description.

invalidation list which contains addresses of the nodes with copies of the page, a pointer to the page table entry of the page, and a pointer to the core page.

A page can be in one of 8 states as shown in Table 1. Initially the state is set to **nothing**. When a page has been allocated then it can be in either the **copy** or **owner** state. If a process has faulted on the page then it is in the **intransit** state. When a page is being invalidated it will be in one of the **owninval**, **transpg** or **transown** states. After a page has been invalidated the state will be **nothing**. The transition between these states is defined further below.

5.2 Server to server communication

The inter-server messages which are used to implement the coherence protocol are described here. There are a number of other messages used in starting up and shutting down a DVSM region, but these are not described. A flow chart showing the main inter-server message types and the subsequent actions of the server are shown in Figure 2. This diagram does not show how invalidation is separated into the sending and waiting for acknowledgements phases, as described below.

5.2.1 Ownership request message

An ownership request contains the region key and page number of the required page, and a pointer to a core page on the requesting node to which the page should be copied. The action of the server depends on the state of the page in the server's page table for the region. If the page is being invalidated or transferred then the request is queued to be processed later. If the page has never been referenced before, then the server returns a *zero grant* message to the requesting server and sets the probable owner field in the page table. If the server is not the owner of the page, then the request is forwarded to the probable owner.

If the server does own the page, then an ownership transfer is initiated. The server first sends an **invalidate** request to the servers in the invalidation list. An invalidate request is not sent to the server requesting ownership if it is in the invalidation list, and therefore has a copy of the page. The core page pointer on the requesting node is also stored in the page table. The server then continues to satisfy other requests until all the *acknowledges* have arrived.

If there are no copies of the page, then it is not necessary to send any invalidations. If the requesting server has a copy of the page then it is also not necessary to transfer a copy before granting ownership. In this case the server executes an *invalidate* kernel call, and then sends an *ownership-grant* message to the requester. If the requester does not have a copy of the page, then the server executes a *copy-invalidate* kernel call, before granting ownership to the requester.

5.2.2 Copy request message

A copy request contains the region key and page number of the required page, and a pointer to a core page as above. If the page is being invalidated or transferred, then the copy request is queued. If the page has never been referenced before, then the server returns a *zero grant* message to the requesting server (this server then becomes the owner even though it only requested a copy). If the server is not the owner of the page, then the copy request is forwarded to the probable owner. If the

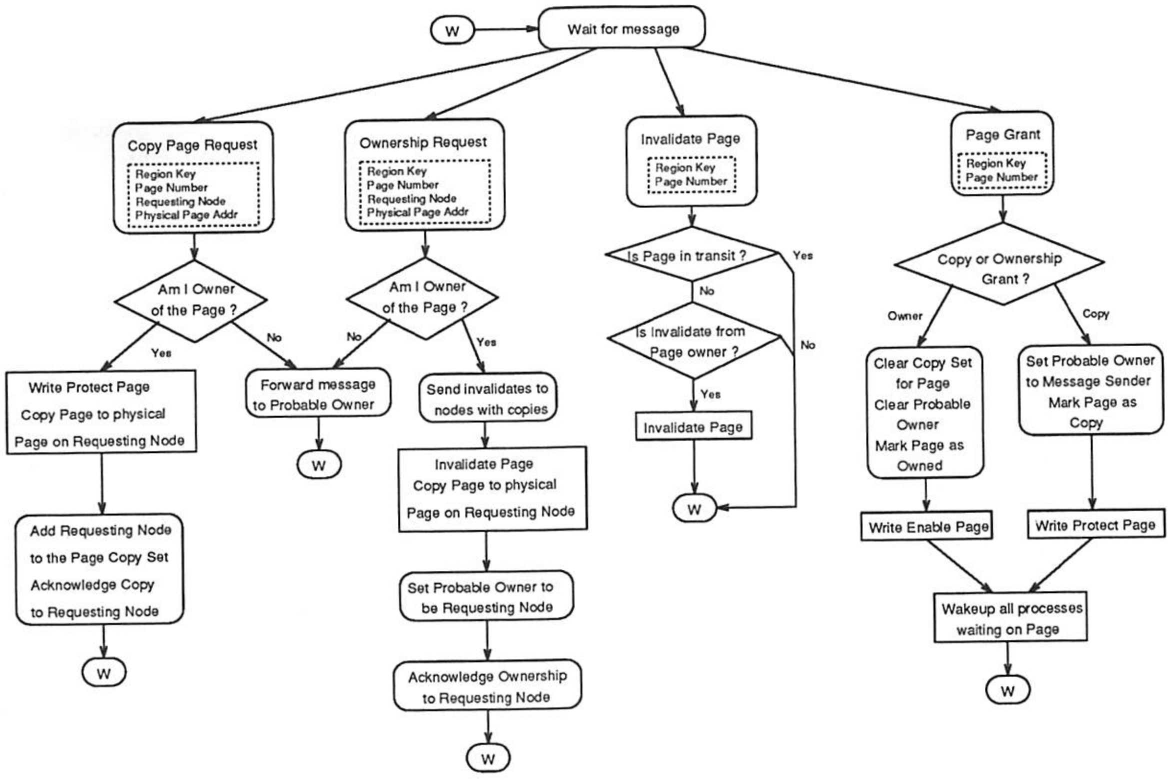

W
Wait for message
Copy Page Request
Region Key
Page Number
Requesting Node
Physical Page Addr
Ownership Request
Region Key
Page Number
Requesting Node
Physical Page Addr
Invalidate Page
Region Key
Page Number
Page Grant
Region Key
Page Number
Am I Owner of the Page ?
Yes
No
Write Protect Page Copy Page to physical Page on Requesting Node
Add Requesting Node to the Page Copy Set Acknowledge Copy to Requesting Node
Am I Owner of the Page ?
No
Yes
Forward message to Probable Owner
Send invalidates to nodes with copies
Invalidate Page Copy Page to physical Page on Requesting Node
Set Probable Owner to be Requesting Node
Acknowledge Ownership to Requesting Node
Is Page in transit ?
Yes
No
Is Invalidate from Page owner ?
No
Yes
Invalidate Page
Copy or Ownership Grant ?
Owner
Copy
Clear Copy Set for Page Clear Probable Owner Mark Page as Owned
Set Probable Owner to Message Sender Mark Page as Copy
Write Enable Page
Write Protect Page
Wakeup all processes waiting on Page

Figure 2: Flow chart showing server actions in response to various inter-server messages (square box indicates kernel call).

server does own the page, then the requesting server is added to the invalidation list and the server executes a *copy-set* kernel call. A *copy-grant* message is then sent to the requesting server.

5.2.3 Ownership, copy and zero grant message

A grant message contains the region key and page number of the page to which access is being granted. The servers actions depend on the type of grant. For an ownership grant the server resets the copy count in the page table, sets the page state to **owner**, and executes the *repair* kernel call to mark the page read-only. For a copy grant the server sets the page state to **copy**, sets the probable owner to be the source of the grant message, and repairs the page read-only. For a zero grant the server resets the copy count in the page table, sets the page state to **owner**, and repairs the new page as writeable.

5.2.4 Invalidate message

An invalidate request contains the region key and number of the page to be invalidated. If a fault on the page is currently being serviced (**intransit**), then the server replies with an *acknowledge* message. If the server has a valid copy of the page, then the server executes the *invalidate* kernel call and sets the page state to **nothing**. An *acknowledge* message is then sent back to the owner[6].

5.2.5 Acknowledge message

An invalidate acknowledge contains the region key and number of the page that has been invalidated. The server waits until all the servers with copies of the page being invalidated have replied with an

[6]In the event that the source of the invalidate request is different from the probable owner then the request is ignored.

acknowledge message. If the owner was invalidating in response to a local write-protect page fault (**owninval**), then the server executes the *repair* kernel call and sets the page state to **owner**. If the owner was invalidating in response to an ownership request and the requester has no copy of the page (**transpg**), then the server executes the *copy-invalidate* kernel call. If the requester does have a copy of the page (**transown**), then the server executes the *invalidate* kernel call. In both cases an *ownership-grant* message is sent to the probable owner. Any requests which have been queued for the page being invalidated are then serviced.

6 Kernel support and organisation

In order to support DVSM some additions were required to the MESHIX kernel, such as the DVSM page fault handler and the DVSM specific system calls, which make up the external-pager interface. Some of the kernel region code and data structures also had to be extended, to include the new DVSM region type.

Unlike the other MESHIX regions the DVSM region spans multiple nodes, and there must be a DVSM region table entry on every node on which the region is accessed. All processes accessing the DVSM region on a node share the same region table entry. Each DVSM region can have it's own server process, which is identified by a new field (the port number of the server process) in the region table entry.

The process table entries have a new field which points to a DVSM region table entry (when the process is attached to one). Two or more processes share a region by setting their DVSM region pointers to point to the same DVSM region table entry. Currently a process can therefore only attach to one DVSM region at a time. It is possible to attach, detach, and then attach to another DVSM region though. Note that a server process does not itself attach to a DVSM region it manages, and so the server does not have access to the DVSM itself.

6.1 Page fault handler

The DVSM region has its own page fault handler routine which communicates with the DVSM server. When a page fault occurs the process traps into the kernel page fault routine, and if the fault occurred in a DVSM region then the DVSM fault handler is executed instead of the standard fault handler. The fault handler executes in the context of the faulted process but with kernel permissions.

On a fault the handler is passed the faulting page table entry (*pte*), address, region, process and access type as parameters. The actions of the fault handler depend on the fault type and the status bits in the *pte*. Three types of page fault can occur for the fault handler to deal with, and are described below.

6.1.1 Unallocated page fault

A reference to a page for which there is no core page allocated. The handler allocates a new core page, marks the *pte* as in transit, and sends a *no page* message to the local DVSM server. The handler then sleeps on the core page address.

6.1.2 In transit page fault

A reference to a page which is marked in transit. A page fault has already occurred on the page and is being serviced, so the fault handler simply sleeps on the core page address. When the page is repaired and the process woken up an immediate fault may occur again, if the earlier fault request did not adequately deal with the fault. This is referred to as the *double-faulting* problem [15].

6.1.3 Read-only page fault

A write has been attempted on a read-only page. The fault handler marks the page's *pte* as in transit, and sends a *write fault* message to the local DVSM server. It then sleeps on the core page address.

6.2 Kernel to server communication

After the fault handler has sent a fault message to the DVSM server and the faulting process is sleeping on the core page address, it is the responsibility of the server to correctly satisfy the fault, repair the page, and wakeup the sleeping process. The actions of the server in response to a fault message are described below.

6.2.1 No page fault message

A no-page fault message from the kernel contains the kernel region pointer, page number, *pte* pointer, core pointer of the faulted page, and the access type. If the page state indicates the page is being invalidated or transferred, then the fault message is queued to be processed later. If the page has never been referenced before, then the server repairs the page writeable and sets the page state to **owner**. If the page state indicates there is no core page allocated (**nothing**), then the server sends an *ownership* request to the probable owner if the access type was write. If the access type was read then a *copy* request is sent to the probable owner. The page state is set to **intransit**.

6.2.2 Write fault message

A write fault message from the kernel contains the kernel region pointer, page number, *pte* pointer and core pointer of the faulted page. If the page is being invalidated or transferred, then the fault message is queued. If the server owns the page, then the server sends an **invalidate** request to all the servers in the invalidation list, and sets the page state to **owninval**. The server then continues to satisfy other requests until all the *acknowledges* have arrived. If the page is a read-only copy, then the server sends an *ownership request* message to the probable owner of the page, and sets the page state to **intransit**.

6.3 Server to kernel communication

The DVSM server requires various functions to be performed by the kernel (since it is a user-level process and does not have access to the kernel data structures). Six DVSM specific system calls have been added to the MESHIX kernel, which can only be called by a process with *root* permissions. The system call library contains a stub for each call that sends the requisite message to the MESHIX *memory manager* process. This kernel-level process then actually executes the call.

6.3.1 Attach system call

The attach call is executed by the server in response to an *attach* request from a process. The parameters are the process to attach the region to, where in the processes address space to attach the region, the region pointer (which is null for a new region), and the region size. If the DVSM region does not exist then it is first created, the server port number is copied into the region table entry, and the region's use count set to 2 (see below). If the region does exist then the use count is incremented. In either case the pointer to the region table entry is copied into the calling processes process table, and the DVSM region is mapped into the processes address space.

Upon creation the use count of the DVSM region is doubly incremented. This prevents the region being automatically destroyed when all attached processes have detached from the region, since the use count does not reach zero. A DVSM region can therefore exist on a node with no processes attached (although this is usually temporary).

6.3.2 Detach system call

The detach call is executed by the server in response to a *detach* request from a process and to finally remove a DVSM region. The parameters are the process to detach the region from and the region pointer. The use count of the region is decremented, and the DVSM region pointer in the process table is cleared. If the use count for the region is zero after decrement then the region is destroyed (the kernel removes the region table entry and reclaims the page tables). The use count after the detach is returned by the call, allowing the server to check whether any processes have died unexpectedly.

6.3.3 Repair system call

The repair call is executed to repair a page's *pte* and wake up any processes sleeping on the page after faulting. The parameters to the call are the page's *pte*, and the values of certain *pte* flags such as the *write-protect* bit. After the *pte* has been marked as *valid*, any processes sleeping on the page are woken up.

6.3.4 Invalidate system call

The invalidate call is executed to invalidate a copy page. The parameter is the page's *pte* and the region pointer. If the *pte* is marked *in transit* then the call returns, otherwise the *pte* is cleared and the core page reclaimed. Pages that are in marked *in transit* cannot be invalidated and reclaimed because the server is expecting to use the page in a subsequent operation.

6.3.5 Copy-invalidate system call

The copy-invalidate call is used in a change of ownership to transfer the contents of a page to another node and invalidate the local page. The parameters are the destination node and core page address, and the source page's *pte* and region pointer. The page is first copied to the other node using a global-copy. If the local page is valid the page's *pte* is then cleared and the page reclaimed. If the page is marked in transit then the page is not invalidated. The transfer is always performed is both cases.

6.3.6 Copy-set system call

The copy-set call is used to copy the contents of a page to another node and write-protect the local page. The parameters are the destination node and core page address, and the source page's *pte* and region pointer. The local page is first made read-only and then the page is transferred using a global-copy. The write-protect occurs before the transfer so that there are no un-caught writes to the page before the transfer is completed.

7 Virtual memory paging

Without virtual memory support the DVSM server would be limited to having regions less than the main memory size of a node, since once a core page was allocated it would remain so until it was invalidated (or the region is destroyed). We decided the simplest approach would be to integrate the paging of DVSM regions with the current MESHIX mechanism for the other region types, which is controlled by the swap manager process.

When the number of free core pages reaches a predefined low level the swap manager selects target pages on an LRU basis. Depending on the region type and whether they are dirty, the pages are then swapped out to disk or simply discarded. Swapped pages are demand-loaded back into memory when a process refers to them again.

Although the swapping is transparent to the DVSM mechanism, some changes were necessary to the DVSM server and page fault handler, as described below. Also when swapping pages from a DVSM region the swap manager must be able to distinguish between a copy and an owned page. Since the DVSM server is a user-level process the swap manager cannot share its data space, and cannot easily determine what it should do with each page, other than by communicating directly with the DVSM server. To avoid this, the swap manager uses the *write-protect* and *copy-on-write* flags in the kernel page table entries of the DVSM region to identify the page. We know that a writeable page must be owned, but a read-only page can be either a copy page, or an owned page with distributed copies. The *copy-on-write* bit, which is not used for DVSM regions, is therefore used to indicate that a page is owned.

7.1 Copy pages

When copy pages are selected by the swap manager they are simply invalidated, and not swapped out to disk. When they are referenced again the local DVSM server must request a new copy from the owner. This strategy was chosen since a network transfer is faster than a (local) disk access, and we only pay a penalty if the page is actually referenced again. The swap disk is likely to be remote from a node, and would therefore require a network transfer when swapping to disk anyway.

When the swap manager has selected a copy page to discard, it clears the *pte* and reclaims the core page. The DVSM server is not notified that the page has been deleted. Any subsequent access by a process to the deleted page results in a *no-page* fault. When the server receives the *no page* message from the fault handler and the page status is **copy**, it knows that the page has been deleted and requests a new copy from the owning node using a *re-copy* request message. The owner then copies the page to the requesting node, but does not add it to the invalidation list again (since it is already on it), and returns a *copy grant* when the transfer is complete. The local server then repairs the page and the faulting process is re-scheduled.

7.2 Owned pages

Owned pages cannot be discarded, since they may be the only valid version of a page, and are therefore swapped out to a disk. When the swap manager has selected an owned page to swap, it marks the page's *pte* as *swapped out*, and initiates the swap out operation. The core page is only reclaimed when the swap has been completed. Again the DVSM server is not informed that the page has been swapped out. If the page is subsequently accessed by a process then a *swapped out* page fault occurs. The page fault handler then allocates a new core page, and initiates the swap in operation by requesting a block transfer of the required page from the swap device. When the page has been successfully transferred, the swap device notifies the swap manager (rather than the DVSM server), which then repairs the page and re-schedules any sleeping processes. The DVSM server is therefore not involved in the swapping of owned pages.

Apart from access by a user process, a swapped out page may also be referred to by the server if it executes the *copy-set* and *copy-invalidate* kernel calls. In this case the page must also be swapped back in before these operations can be completed.

7.3 Implications

Since DVSM regions are shared, they can be expected to be much larger than a normal data region for any one process. One proposed use of the DVSM is as a distributed cache for database programs. We can expect the DVSM region to be much larger than the total core memory on any one node, and probably larger than the swap space allocated to any one node. Therefore other methods of swapping will be required for large DVSM regions. Possible solutions are to have one or more disks allocated solely for swapping DVSM, or to use free core pages on other nodes for swap space (see Section 10).

As a performance enhancement, when an owned page has been swapped out but is required for an ownership transfer, then the page could be swapped in directly to the new owner, rather than first swapped in to the old owner and then transferred to the new owner. The old owner would still have to send a *grant* message to the new owner.

8 Synchronisation within DVSM

The only synchronisation mechanism currently available with the DVSM is a spin-lock function. To obtain exclusive access to a shared data structure, a process must first lock an associated binary semaphore. The process then performs some operations on the shared data, and unlocks the semaphore. Any other processes contending for the lock will busy-wait until it is available. Because more than one process on a node may be contending for the lock it is necessary to either turn interrupts off or use an indivisible instruction to set the locks. We use the MC68030 test-and-set instruction which first reads the given memory location, and then sets a bit in it. If the returned value is zero then the lock was free, and will now be set. If it was locked then a non-zero value is returned, but the value is unchanged. If the page is write protected then a write fault is always generated by the test-and-set, whether or not it succeeded.

8.1 Implementation

In the lock function we first test the value of the semaphore. If it is non-zero, which means the lock is held, then we loop. Otherwise the test-and-set is executed. If the result of the test-and-set is non-zero, then another process has gained the lock first, and we loop again. Otherwise we now have the lock. The unlock function zeroes the semaphore.

The initial test of the semaphore in the lock function is an efficiency measure. If the page that contains the semaphore is not in memory then a no-page fault occurs, and a read-only copy will be delivered. If the semaphore is locked then the process will spin on this first loop, without generating write faults. When the lock is released the page is invalidated by the owner, and the spinning process faults again. When the updated page is delivered, the first process to get through the test-and-set will get the lock, again having to invalidate any other copies. Without the first test the process would be continually generating write faults, and there would be a lot of thrashing. Now there is only a lot of activity when the lock is actually set and when it is released.

8.2 Implications

Contention for locks can generate a large amount of network traffic, and spinning on locks is not efficient. Also, locks may be placed on the same page as other data structures, access to which might generate interference. Ideally we would want the processes to be queued on the lock (sleeping), and then be woken up in order as the lock is released. A number of methods have been used in order to make synchronisation in DVSM systems more efficient, for example by not using the DVSM at all (see Section 10).

9 Performance of DVSM

Ideally we would like to minimise the delay in delivering a page after a fault as much as possible. This is limited by the message latency of the network hardware, and the software overhead in the operating system. Assuming these are fixed, some optimisations can be made to the number of message that have to be transferred in the coherence protocol, and to the length of probable owner chains.

We have carried out some preliminary performance testing of the DVSM implementation. The basic page fault times and the speed-up of a matrix multiply program have been measured[7]. In these measurements as much of the other system activity as possible was minimised, and local timings on each node were made using a micro-second clock.

9.1 Page fault times

The fault time is measured from when the faulting process enters the fault handler routine to when it exits the routine, and therefore represents the page fault time as seen by the process. Each measurement is the average for 100 faults.

The local zero-page fault time is $2014\mu s$, and includes the time to allocate a new core page. The timings for a write fault (ownership request) and read fault (copy request) between two nodes only,

[7]Because MESHIX did not include a remote fork operation we have been unable to port other benchmark programs, such as the SPLASH suite [16]. Work is currently in progress to add such a facility.

Internode distance	Write fault time (μs)	Read fault time (μs)
1	8158	8109
2	8143	8127
3	8136	8139
4	8278	8117

Table 2: Representative page fault times for DVSM in micro-seconds, with increasing distance between faulting nodes.

Nodes	Execution time (s)	Relative speed-up
1	125	1.0
2	66	1.9
4	37	3.4
8	22	5.7

Table 3: Execution time in seconds and relative speed-up for parallel matrix multiply with 1 to 8 nodes.

with increasing distance between the nodes, are shown in Table 2. These results were collected on an 8-node machine. The table shows that ownership and copy requests have similar fault times (since they both require the same number of messages, and there are no invalidations for the ownership transfer). The increasing internode distance does not appear to affect fault times, since only the connection time and not the transfer bandwidth changes with internode distance. These results can be compared to an average page fault time of about $4000\mu s$ for the Shiva DVSM system [6], and $3000\mu s$ for the KOAN system [7], both implemented on an iPSC/2 hypercube (see Section 10).

9.2 Matrix multiply

The parallel matrix multiply program multiplies two integer matrices which are placed, with the result matrix, in a DVSM region. The program is divided into a father and a number child processes. The father process first creates the DVSM region, reads the matrix data from a file and writes it to the DVSM region. The process then uses fork to create a given number of child processes which exec the child code and are migrated to another node in doing so. The father process then wait's for the child processes to exit.

Each child process first attaches to the DVSM region, and then starts doing it's part of the matrix multiply. The result matrix is divided into n horizontal strips for n children, and each child process calculates the entries for it's strip. When all the children have finished calculating their part of the matrix and exit, the father process reads the result matrix from the DVSM region and writes it to a file.

9.2.1 Speed-up

To see if any speed-up was achieved by increasing the number of child processes executing in parallel, the matrix multiply was executed using 1 to 8 child processes on an 8 node machine, with 256×256 input matrices. The resulting execution time in seconds and relative speed-up are shown in Table 3, and show that we do indeed achieve some speed-up by parallelising the matrix multiply. The execution time was measured in the father process from before the fork to after all the child processes exit, and the wait returns. These results do not therefore include the file access time, which remains constant as the number of child processes increases.

Notice that the matrix size has been deliberately chosen so that the matrices are aligned on the page boundaries to minimise contention for shared pages. In more realistic programs, especially those with shared locks, there is likely to be far greater contention and resultant thrashing.

9.2.2 Page fault times

The page fault times given above were for an unloaded network. The page faults time in the execution of the matrix multiply program on 8 nodes were measured and shows that page fault time can increase dramatically, the maximum measured was about $20000\mu s$, when an application is executing and the network is loaded. Some of this latency is due to the back-off time between re-tries when there is a network routing conflict, which can be up to $1666\mu s$.

9.3 Improving performance

Because ownership may have changed an ownership or copy request may have to be forwarded a number of times before it reaches the current owner. A possible optimisation is to allow copy requests to be serviced by an intermediate server that has a valid copy of the page, rather than by the owner [3]. The copy set thus becomes distributed around the servers, in a tree with the owner as the root. When the owner sends out invalidations, these are then propagated by each intermediate server to the others in their copy set. This can reduce the time required for invalidation (from n to $\log n$ with n copies), and also the time required to find a valid copy.

In an ownership transfer it is necessary to send a *grant* message after the global-copy has been completed to inform the new owner of this. If the global-copy routine was changed to allow a completion message to be sent to the new owner, then an additional *grant* message would not be necessary. This would require re-definition of some of the kernel network primitives.

10 Comparison to other implementations

Our implementation has similarities to the IVY, SHIVA and KOAN systems. The dynamic distributed manager algorithm was developed and compared to other algorithms on IVY. Both SHIVA and KOAN are DVSM systems implemented on distributed memory multiprocessors, but use a fixed manager algorithm. They also both have a mechanism to allow pages to be swapped to other nodes rather than a swap device. IVY used the DVSM for synchronisation, but both SHIVA and KOAN use seperate mechanisms. A summary of these systems is given below.

IVY is a DVSM programming environment for a network of Apollo workstations [17]. IVY provides a thread based library with procedures for process control and memory management. A user-level server executing on top of the Aegis operating system maintains strict coherence of the shared memory using a dynamic distributed page ownership strategy, and broadcast write-invalidation. The MMU on each Apollo catches page faults which are then passed to server. Synchronisation is accomplished using explicit locking of the shared memory and event count structures.

SHIVA [6] is DVSM programming environment on an iPSC/2 hypercube containing 80386 processors (up to 128 nodes). Fixed distributed servers maintain strict coherence using write-invalidation, and run on top of a small message-passing kernel on each node. Swapping of shared memory pages onto backing store and other nodes is supported. Semaphores are implemented using a distributed data structure on each node and message-passing (the semaphore data structure is not in the shared memory). The semaphore migrates around nodes, and a mechanism similar to dynamic distributed server is used to find the semaphore.

KOAN [7] is a DVSM implementation on an iPSC/2 hypercube. The DVSM system interfaces to the NX/2 kernel on each node, and no other thread and memory management facilities are supplied. A fixed distributed ownership strategy with write-invalidation is used to maintain strict coherence. Swapping of pages to other nodes is supported. A seperate token-passing synchronisation mechanism is used.

11 Conclusion and further work

We have described, in great detail, the implementation of a DVSM system on the Topsy multicomputer. This implementation is based on user-level server processes distributed across the machine, which maintain coherence of the paged DVSM region using invalidation [3]. The DVSM servers communicate with the operating system using an external-pager interface, which was added to MESHIX Simple spin-lock synchronisation primitives are provided for synchronisation within the DVSM, and virtual memory paging of the DVSM region was integrated into the existing MESHIX paging mechanism. A preliminary performance evaluation showed that page fault time is relatively poor, mainly due to the reduced network bandwidth. A parallel matrix multiply program using the DVSM showed some speed-up when executed on up to 8 processors. The implementation was hampered because we has to add an external-pager interface to MESHIX first.

Further development to the DVSM system is likely to be application driven. Colleagues have expressed interest in porting some applications onto the DVSM, such as a parallel Parlog system (JAM [18]) and the Postgres database system. This will require an rfork system call to be implemented, and a System V shmem interface to the DVSM (plus an implementation of the other System V IPC facilities). Performance improvement will be focussed mainly on the TOPSY hardware, although reducing the number of messages transferred in the coherence protocol, and allowing the option of weaker coherence, are also goals. Substantial gains could also be achieved by making the DVSM server a kernel-level process, in which case execution of system calls and manipulation of page tables could be carried out directly and would not involve the memory manager process. This is contrary to the micro-kernel philosophy though, and could create synchronisation problems within the region handling code. The DVSM implementation has led us on to investigate whether DVSM could be used as the basis for an operating system system based on a single coherent shared address space [19].

12 Acknowledgements

TOPSY and MESHIX were written and directed by Phil Winterbottom and Tim Wilkinson, and produced by Peter Osmon. Thanks to Paul Kelly for pointing out that synchronous invalidation was necessary for correct operation, and Martin Cripps for suggesting the correct solution to a bug in the copy-invalidate routine.

The research described here was supported by the United Kingdom Science and Engineering Research Council and the Department of Trade and Industry.

References

[1] P. Winterbottom and P. Osmon, "Topsy: an extensible UNIX multicomputer," in *Proceedings of UK IT90 Conference*, (Southampton University), pp. 164–176, March 1990.

[2] P. Winterbottom and T. Wilkinson, "MESHIX: a UNIX like operating system for distributed machines," in *UKUUG Summer Conference Proceedings*, pp. 237–246, July 1990.

[3] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, November 1989.

[4] M.-C. Tam, J. Smith, and D. Farber, "A taxonomy-based comparison of several distributed shared memory systems," *ACM Operating Systems Review*, vol. 24, pp. 40–67, July 1990.

[5] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *IEEE Computer*, vol. 24, August 1991.

[6] K. Li and R. Schaefer, "A Hypercube Shared Virtual Memory System," in *Proceedings of the International Conference on Parallel Processing*, pp. 125–132, 1989.

[7] Z. Lahjomri and T. Priol, "KOAN: a Shared Virtual Memory for the iPSC/2 hypercube," Tech. Rep. 597, IRISA, July 1991.

[8] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, implementation, and performance evaluation of a distributed shared memory server for Mach," Tech. Rep. CMU-CS-88-165, Carnegie-Mellon University Computer Science Department, August 1988.

[9] T. Wilkinson, A. Whitcroft, P. Winterbottom, and P. Osmon, "The Meshnet Multi-stage Communications Network," Tech. Rep., City University Computer Science Department, May 1991.

[10] T. Stiemerling, A. Whitcroft, T. Wilkinson, and N. Williams, "Evaluating Meshix — a Unix compatible micro-kernel OS," Tech. Rep., City University Computer Science Department, January 1992.

[11] M. Bach, *The Design of the UNIX Operating System.* Prentice Hall, Inc, 1986.

[12] V. Abrossimov, M. Rozier, and M. Gien, "Virtual Memory Management in Chorus," Tech. Rep. CS/TR-89-30.1, Chorus systémes, May 1989.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 15–26, 1990.

[14] *UNIX System V Release 3.* AT&T, 1986.

[15] R. Kessler and M. Livny, "An analysis of distributed shared memory algorithms," in *Proceedings of the 1989 International Conference on Distributed Computing Systems*, pp. 498–505, 1989.

[16] J. Singh, W. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-memory," Tech. Rep., Computer Systems Laboratory, Stanford University, 1991.

[17] K. Li, "IVY: a shared virtual memory system for parallel computing," in *Proceedings of the International Conference on Parallel Processing*, pp. 94–101, August 1988.

[18] J. Crammond, "The Abstract Machine and Implementation of Parallel Parlog," Tech. Rep., Department of Computing, Imperial College, July 1990.

[19] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly, "Angel: a proposed multiprocessor operating system," in *European Workshops on Parallel Computing 92*, March 1992.

Integrating Consistency Control and Distributed Shared Memory: The Travails of an Implementation*

Raymond C. Chen
(chen@sni-usa.com)
Siemens-Nixdorf Information Systems
Research and Development Division
4 Cambridge Center
Cambridge, MA 02142

Partha Dasgupta
(partha@enuxva.eas.asu.edu)
Department of Computer Science
and Engineering
Arizona State University
Tempe, AZ 85287-5406

Abstract

Distributed Shared Memory (DSM) is a mechanism that allows networked computers to share memory, a feat that is normally prohibited in the definition of distribution. The DSM technique of demand paging memory pages across a network and providing coherence between processes concurrently accessing memory has promised to make distributed systems almost equivalent to shared-memory multiprocessors. This paper is about the traps that lurk behind this illusion.

Consistency control mechanisms enable users to write and execute programs that transform a system from one consistent state to another consistent state. The results in this paper stem from an effort to implement a consistency control mechanism with transaction-like features called IBCC in a general-purpose distributed system that is built around DSM. This paper describes how DSM and consistency control can be integrated: the functionality that must be integrated into DSM and the functionality that may be best left out of DSM to be handled by a higher-level, orthogonal layer. This paper explains why DSM and consistency control cannot be simply integrated as two layered services, describes the problems that DSM presents to a higher-level service that implements transactions and shows what modifications are necessary to a generic DSM scheme to support transactions and and what modifications should *not* be made to DSM.

1 Introduction

Distributed Shared Memory (DSM) is a mechanism that allows networked computers to share memory, a feat that is prohibited in the definition of distribution.[1] The technique of demand paging memory pages across a network and providing coherence between processes that concurrently access memory has promised to make distributed systems almost equivalent to shared-memory multiprocessors. This paper is about the traps that lurk behind this illusion.

*This work was supported in part by NSF grant CCR-86-19886.
[1] Distributed systems are systems that do not physically share memory.

The results in this paper stem from an effort to implement transaction-like features in a general-purpose distributed system that is built around DSM. DSM is used to manage all the data in the system as one coherent single-level store. Processes can run anywhere, access the data, modify it and send it back to the DSM server. The DSM server keeps a copy of the memory on disk, providing long-term persistent and stable storage. Thus the DSM server is like a shared, remotely accessible disk. Transaction systems have long been implemented on top of disk drivers, and hence implementing one on top of a DSM server seemed attractive and simple.

Transactions are a form of consistency control. Consistency control mechanisms enable users to write and execute programs that transform a system from one consistent state to another consistent state. In 1987, we formulated the basic semantics of a set of consistency control mechanisms for object-based environments that we call Invocation-Based Consistency Control (IBCC). For the purposes of this paper, IBCC can be considered to be a transaction facility for shared memory computations. IBCC manages the consistency of data stored in addressable memory by creating versions when transactions execute and committing them to storage when the transactions complete. By 1988, we had refined the semantics of IBCC and sketched out a design that was intended to be implemented on the *Clouds* distributed operating system [Spa86, PD88].

At that point *Clouds* was in the process of being re-designed and re-implemented as *Clouds v.2*. *Clouds v.2* went into operation in 1989 and had a microkernel supporting the operating system [DRA91, DCM+90]. Among other changes, one fundamental aspect that changed was that *Clouds v.2* incorporated DSM for storing all user-level data and programs. DSM made *Clouds v.2* a very appealing system. All resources (e.g. objects) were available on all sites.

1.1 DSM and RPC

Invoking a remote object on a system that does not have a DSM mechanism (a non-DSM system) requires performing a *Remote Procedure Call* (RPC). On a non-DSM system, when a program invokes an object that is not physically present on the local site, the operating system must marshal, transmit and unmarshal the RPC arguments, spawn a slave process on the remote site, duplicate the computation control information and environment on the remote site for the remote process, execute the object invocation on the remote site and *then* marshal and transmit the return results back to the first site as well as clean up the remote process and its computation control information. However, in a system where objects are managed by a DSM system, all objects can be accessed from any site as if the objects were present locally. Therefore, in a DSM system the object is simply mapped into the address space of the process and when a page that is not present on the local machine is referenced, the DSM mechanism locates that page and pages it in across the network. Thus DSM, even more so than RPC, simplifies distributed computing by presenting the illusion that all computation is local.

Furthermore, the existence of a DSM mechanism makes it easier to implement other desirable features. A DSM mechanism allows easy migration of objects and computation almost automatically (and almost "for free"), from one machine to another machine. Especially powerful is the concept of implementing *RPC on top of DSM*. Using an RPC, an object can be invoked anywhere on the system as long as the object is handled by DSM. That is, if a DSM-controlled object O is invoked at site S, and O is not located on site S, O will be delivered to S via DSM. This feature makes it possible to distribute a computation by transforming object invocations into RPC requests that can be sent to any machine regardless of where the data (object) actually resides. DSM will ensure that the data is moved to the right place when needed. We call this type of RPC a "DSM-RPC" since the RPC mechanism depends upon the DSM mechanism for the RPC to work.

DSM-RPC makes it possible to easily load-balance on object invocations. If a process executing on a heavily loaded machine invokes an object, that invocation can be run on another machine by using a DSM-RPC. Given a system configuration where there exists a mix of powerful computation engines, data (object storage) servers, and personal workstations, the presence of a DSM mechanism allows the most effective use of each type of machine with the majority of the computation occurring on the computation engines, objects being stored on the data servers, and the user interface running on the workstations.

DSM is the ultimate panacea for distributed computing. With DSM, distributed programming looks like centralized (or parallel) programming. The presence of DSM can simplify the design and implementation of very desirable features. All these advantages of DSM led us to overlook some of the hidden problems of working with DSM — until we ran up against them while integrating IBCC and DSM.

Since *Clouds v.2* supported DSM as its primary means of permanent storage, we decided to implement IBCC on top of DSM. All objects controlled by the IBCC scheme would reside with the DSM servers and would be available at all hosts. Since a DSM controlled storage device looks to a *Clouds* site like a normal local storage device, we hoped that consistency control and DSM were *relatively orthogonal* and that our initial implementation design could be extended in a simple and straight-forward fashion to work with DSM. In addition, we hoped that the locking necessary for transactions could be provided by the DSM system (which in our case was already built in).

Our orthogonality argument was simple. IBCC was designed to use local disks as the storage medium. DSM could also be used as the storage medium as we could write back the segments to the DSM server as if it were a local disk. Thus an implementation of IBCC should be the same on top of DSM as it would be on top of a disk driver.

The locking argument was even simpler. Since DSM managed the data, it could lock pages of the data as it provided them and unlock them as they were written back.

1.2 The Surprises

When we started working on the implementation we received a rude shock. In fact it was one surprise after another. The first surprise was that the DSM supported locking could not be used for transactions. The subsequent series of surprises involved discovering that we were absolutely wrong, on many counts, about the orthogonality of DSM and transaction management. In general, we were wrong in assuming that a distributed system with DSM support works like a shared-memory multiprocessor and hence about the simplicity of implementing anything on top of DSM.

Integrating consistency control and DSM in an object-based environment turned out to be a major task. We had to make significant changes to the DSM design and implementation as well as changes to the IBCC design. In fact, DSM had to be enhanced to support functionality which we had felt should not be a part of DSM at all and stripped of functionality that was already a part of DSM. Although some of the necessary functionality can be implemented in an orthogonal fashion, and some of the functionality *should* be (and was) implemented in an orthogonal fashion, critical functionality had to be moved into DSM. This intertwining of DSM with IBCC made the implementation quite non-orthogonal.

The semantics of IBCC are powerful and unfortunately too involved to discuss in full in this paper. Interested readers are referred to [CD89, Che91, CD91]. However, an understanding of IBCC semantics is not necessary for the purposes of this paper. As mentioned earlier, when used in a simple fashion, IBCC can be thought of as a transaction facility and the majority of the problems presented in this paper apply when integrating transactions with DSM. In fact they would apply when integrating any pessimistic consistency control mechanism with DSM in a distributed system. There was only one instance where the differences between IBCC semantics and transaction semantics caused a problem that might not otherwise have occurred. This is the result of IBCC allowing transactions and processes to coexist and cooperate. However, that one instance pointed out a more general problem with DSM which is why we will mention it in section 5.2. Therefore, except for that one instance, for the sake of clarity, we will use transactions and transaction semantics to illustrate the problems we encountered in integrating consistency control in the form of IBCC with DSM.

This paper describes how DSM and consistency control can be integrated: the functionality that must be integrated into DSM and the functionality that may be best left out of DSM to be handled by a higher-level, orthogonal layer. This paper explains why DSM and consistency control cannot be simply integrated as two layered services, describes the problems that DSM presents to a higher-level service that implements transactions and shows what modifications are necessary to a generic DSM scheme to support transactions and what modifications should *not* be made to DSM.

Most of these problems are a result of the fact that in a non-DSM environment, one can assume that if an object resides on a site, data in that object can be accessed only by computations executing on that same site. We call this assumption the *locality assumption.* However, in a DSM environment, the locality assumption does not hold since the primary goal of DSM is to allow data on one site to be accessed by computations running on any site in the distributed system.

The remainder of this paper addresses the problems we encountered integrating a consistency control mechanism with a DSM mechanism. We first present a brief overview of distributed shared memory, introduce the concept of a persistent object environment that uses a single-level store, and then discuss consistency and transactions. We then present the problems we encountered when integrating consistency control with DSM and our solutions to those problems.

2 Distributed Shared Memory

In 1986, Li and Hudak introduced the concept of distributed shared memory [LH86]. Distributed shared memory allows loosely-coupled distributed system to share a global virtual address space. In Li and Hudak's scheme, the shared virtual address space supports *single-copy shared* or *strongly coherent* semantics. That is, although the memory may be physically spread out among many sites in the distributed system, there appears to be only one copy of the memory which is shared by all sites in the system.

Li-Hudak DSM is usually implemented using a variant of a multi-processor cache coherence scheme based on invalidation (as opposed to a write-back/snoopy coherence scheme). Each page can be thought of has having an "owner" that controls access to the page and "keepers" that possess copies of the page. A page may have multiple keepers (say in the case of a read-only text page being used by multiple sites), but only one owner.

If a process requires a page but does not have an accessible copy, a page-fault occurs. The page-fault handler sends a request to the owner of the page. If the page-fault is a read fault, the owner forwards the latest copy of the page to the requesting site and adds the site to its list of keepers for that page. If the page-fault is a write fault, the owner *invalidates* all outstanding copies of the page by sending invalidation messages to all keepers of the page, and forwards the latest copy of the page to the requesting site.

As in any invalidation-based cache coherency scheme, the page can shuttle or thrash between sites if the same page is heavily used simultaneously by two different sites. However, parallel applications that use DSM should be written to avoid this *memory contention* so hopefully, thrashing will not occur very often in a Li-Hudak DSM system.

However, a number of schemes have been proposed to minimize thrashing and invalidation messages. In [FP89], Fleisch proposes a DSM scheme where a page can be pinned (held) on a site for some arbitrary interval of time, similar to the way a cpu can be held by a process for some interval of time in a time-slicing, multi-tasking operating system. If another site requests the page from the current keeper site and the time interval has not yet expired, the keeper keeps the page until the time interval expires. The time interval for each page can be a fixed constant (static) or tunable (dynamic).

Another technique of DSM service proposed by Ramachandran and Khalidi integrates DSM with synchronization [RAK89, Kah89]. Sites can request pages in either *read*-mode, *read/write*-mode, *none*-mode, or *weak-read*-mode. Like Li-Hudak DSM, there is the concept of an owner and keepers. However, unlike Li-Hudak DSM, read-mode and read/write-mode involve page-level locking.

The keeper of a page in *read*-mode has shared read access to the page *with* the guarantee that the page will not be yanked away (invalidated) unless and until the keeper explicitly discards (or unlocks) the page. The keeper of a page in *read/write*-mode has *exclusive* access to the page with the guarantee that the page will not be invalidated unless and until the keeper explicitly discards the page. The keeper of a page in *none*-mode has exclusive and coherent access to the page but the page is liable to be invalidated at any point if another site accesses the page. The keeper of a page in *weak-read*-mode has non-exclusive access to the page with *no* guarantee as to the coherence of the page.

Read and read/write modes are meant to be used by programs that perform data locking. When a site reads (or writes) a page, the owner receives a request for the page and if possible, sends the page to the site (making that site a keeper of the page) and sets a read-lock (or write-lock) on the page. When the keeper discards the page, the owner unlocks the page. Further requests for that page by other sites with conflicting lock modes are kept pending until the lock can be granted. None mode is functionally equivalent to the write mode of Li-Hudak DSM.

Read and read-write modes seem advantageous if processes lock the data they use. Otherwise, the system has to guess when to unlock/discard pages. If the system guesses wrong, the behavior is unpredictable and includes inconsistent execution as well as a loss of concurrency. Thus the locking modes in DSM are not usable unless the processes explicitly lock and unlock the shared pages they use.

However, when transactions use the DSM pages to access data, integrating locking with DSM service seems to make perfect sense. Transactions always lock and unlock data they use. However in section 4.2, we show why integrating locking and DSM cannot be done using the simplistic mechanism (page modes) proposed by Ramachandran and Khalidi.

2.1 *Clouds* and DSM

The *Clouds v.2* DSM integrates synchronization with DSM service as proposed by Ramachandran and Khalidi. However, at the time of the IBCC implementation, the majority of the programs written for the *Clouds* system did not perform explicit page-level locking. Text pages (program code) were marked read-only, therefore those pages were used in *read*-mode. However, virtually all the *Clouds* applications used shared data pages in *none*-mode. This made *Clouds* DSM appear to be functionally equivalent to Li-Hudak DSM (except that data pages were not replicated even if they were only being read). Our effort was the first system in *Clouds* that performed explicit data locking and was hence expected to use the locking modes that *Clouds* DSM provided. We did not do so. *Why* we did not do so is described later in this paper.

The *Clouds v.2* DSM uses a centralized server. The server is the owner of the pages and also stores the data. The owner does not change. The owner hands out copies of the pages to keepers, maintains locking information and invalidates pages when necessary. The server is implemented as a single-threaded process on Unix, while the keepers are sites running *Clouds*.

2.2 DSM, Persistent Objects and a Single-Level Store

The DSM system in *Clouds* controls the memory space of large-grained objects. Large-grained objects are large protected address spaces (size is conceptually limited only by the address space of the machine) containing code and data. Data inside the object is accessible only by the code inside the object. Objects are *persistent.* Once created, objects exist until explicitly destroyed. Objects are also *sharable.* That is, multiple threads of execution may execute concurrently inside the same object and all data in the object's address space is accessible by all threads. Of course, threads move from one object to another object through object invocation.

Thus, while the union of all objects in the system controlled by DSM may be thought of as a large, shared virtual address space, DSM actually manages multiple pages in multiple address spaces. A single address space (an object) may be used at multiple sites, by multiple threads of execution at each site. The DSM service stores pages on disk for long-term persistence. Thus each object is a single-level store. The IBCC system is designed to provide consistency among the single-level stores.

3 Consistency Control using Transactions

One particular form of consistency control is transactions. As mentioned earlier, though IBCC is not a transaction facility, we will equate consistency control with transactions for our discussion. To provide for consistency, a pessimistic consistency control mechanism, such as transactions, must satisfy the following requirements:

- The mechanism must be able to perform concurrency control on the data touched by consistent computations.
- The mechanism must be able to preserve the older state of persistent data so that changes to persistent data may be undone.
- The mechanism must allow a consistent computation to atomically commit changes made to persistent data.

- The mechanism must allow a consistent computation to atomically abort changes made to persistent data.

A transaction management system detects reads from and writes to persistent data by the transactions. The accessed data is locked in the appropriate mode and if the access is a write, the state of the persistent data is saved before the write is allowed to proceed. The changes made by a transaction are then typically committed or aborted when the transaction terminates and the locks are released. The commit (or abort) is atomic and is usually two-phase if a transaction makes data changes on multiple sites. Thus, transactions never work directly with persistent data. Instead, they work with intermediate (or shadow) versions of the data. This is especially true for nested transactions where a child may update data and abort and then the parent uses the older version of the data. Changes made by transactions to these intermediate versions are either atomically committed (replacing the previous state of the persistent data) or atomically aborted (leaving the persistent data unaltered by the effects of the transaction). Transactions also perform two-phase read/write locking on data, ensuring that the effects of concurrently executing transactions are *serializable.* Serializability is a form of consistency that ensures that no anomalies will occur as a result of concurrently-executing transactions that read and/or write the same data items.

The four requirements mentioned earlier are the heart of consistency control. The first requirement guarantees that computations have *view atomicity* — they are atomic with respect to other concurrently-executing consistent computations. The other three requirements ensure *failure atomicity* — that computations are atomic with respect to failure. However, as will be seen, simply layering consistency control on top of DSM can cause each of the four requirements — concurrency control, version preservation, commits, and aborts — to fail.

3.1 Assumptions

The rest of the paper makes the following assumptions.

- Computations run on machines on a network. Some of these computations are transactions.
- Transactions access data items that reside on a DSM server. The data is paged in across the network. When a data item is accessed by a transaction, it is locked.
- After the transaction is complete, it writes back the updated data to the DSM server atomically (using 2-phase commit) and unlocks the data.
- The transaction may execute on sites other than where it was started. This happens when a transaction performs an object invocation and RPC is used to run the transaction on a different node (why this is done is of no consequence to this paper).
- A transaction can touch multiple data items. A transaction can touch the same data item on multiple nodes.
- Transactions may be nested[Mos81].

In the remainder of the paper, we will discuss the interference between DSM and transaction management and we will present our solutions to the problems.

4 Concurrency Control

A local concurrency control mechanism only controls concurrency among locally-executing threads of control. A distributed concurrency control mechanism controls concurrency among threads of control that may be executing anywhere in the distributed system. Distributed concurrency control requires inter-site communication and is therefore more costly than local concurrency control which requires no inter-site communication.

In a distributed non-DSM environment, consistency-related concurrency control (such as transaction read/write locking) is local. System designers can make the assumption that the only computations that alter data are those that are executing where the data is located. That assumption implies that all concurrency control (lock) requests on data will originate only on the site where the data is located. Therefore, local concurrency control is sufficient to ensure consistency.

4.1 DSM and Locking

Adding DSM to consistency control will cause concurrency control to fail unless the concurrency control mechanism performs distributed concurrency control either though a distributed lock manager or via DSM-supported locking. For example, a transaction typically write-locks data before the transaction begins to alter the data. This write-lock is an exclusive lock. The lock cannot be granted to the transaction if any other transaction holds a lock on the data and once the write-lock is granted, no other transaction may gain a new lock on the data until the write-lock is released.

In a non-DSM environment, the transaction will gain the lock from the lock manager on that site and proceed. If a second transaction attempts to work on the same data, the second transaction will also issue a lock request to the lock manager on that site. Since the lock manager knows that the first transaction already holds the desired write-lock, the lock manager will block the second transaction until the first transaction releases the lock. Thus, consistency is maintained.

However, in a DSM environment, these two transactions may be running on two different sites. The transactions will issue their requests to lock managers on two different sites. If the lock managers are purely local lock managers, *both* lock managers will grant write-locks since the second lock manager will not be aware that the first lock manager has already granted the write-lock. Therefore, *both* transactions will proceed, violating consistency.

In order for consistency to hold, the locking must be distributed locking. This requirement generalizes to any concurrency control mechanism required by a consistency control scheme. If the consistency control mechanisms require that concurrency among executing threads be controlled in some fashion, once DSM is introduced, the locality assumption no longer holds and distributed concurrency control mechanisms must be used or consistency will be violated.

4.2 Distributed Locking Solutions

There are two fundamentally different approaches to providing distributed locking. The first approach is to change the locking mechanism from a local locking mechanism to a distributed locking mechanism. This change is not only non-trivial but also makes the transactions more inefficient especially if they do fine grained locking. On a Sun 3/60 workstation, once the lock manager gains control, local locking can be performed in a small fraction of a millisecond. Distributed locking can add several milliseconds of additional overhead (7 msec in our case) if the local kernel must communicate with one or more other sites.

However a second solution proposed by Ramachandran and Khalidi seems very appealing: integrate synchronization and data coherence by making the lock on a DSM-controlled data item a part of that data item. That is, the lock should be obtained when the data is obtained and the locking is handled by the DSM mechanism. The lock moves with the item and possession of the item is evidence of *possession* of the lock. For all later lock requests originating at the same site, locking on that item can be handled locally.[2] The problem with the above seemingly elegant solution is that it does not work unless *major* changes are made to DSM. The solution breaks down in many ways. A few examples follow:

- DSM manages pages and locks on a per-machine basis [RAK89]. Suppose transaction T modifies data D write-locking the page(s) of D on site S1 and then T later executes on site S2 (via an RPC, for example) and touches D on S2. As far as the DSM mechanism is concerned, the lock is held by a different machine, therefore T will block and deadlock with itself attempting to touch D on S2.

- DSM can be modified to lock on a per-process basis, but that does not work either with distributed transactions since the same transaction will use multiple processes — at least one per site. So the previously mentioned problem still applies. To solve this, DSM has to be able to distinguish between transactions and processes, which breaks the orthogonality of DSM and transaction management.

- DSM is used by *distributed* processes (which we call s-threads) as well as distributed transactions. However, in IBCC, s-threads never block as a result of consistency-related locks. They *always* access the latest version of the data, even if that version is an uncommitted version generated by a transaction.[3] DSM must be able to distinguish between the two and not treat all page requests as lock requests. Otherwise, s-threads will block unnecessarily. Again the orthogonality falls apart.

- Even if DSM is modified to keep track of which *s-threads* and which *transactions* hold a page in addition to which *machine* holds that page, DSM is designed to manage and hence lock *pages* of data while transactions often obtain logical locks that may pertain to data of finer grain or larger grain than a page. Thus logical locking semantics have to be implemented in the DSM server making DSM a very involved system. For example if two data items D1 and D2 are located on the same page, page-level locking will not allow two transactions on *the same site* to access D1 and D2 concurrently, while a lock manager would allow such behavior.

Each of the above problems can be fixed but there are other problems as well. They all cause significant complexity additions to DSM. The simplistic page-level per-site (or per-process) locking cannot be used by transactions. As we delved into the exact means of supporting IBCC locking semantics using DSM, we found that a DSM lock manager looked more and more complicated. In order to provide useful locking support to higher-level layers, DSM must support the exact locking semantics desired by the higher-level layers.

[2] This can lead to starvation, but this problem is not addressed.

[3] The justification for these semantics is beyond the scope of this paper. Interested readers are referred to [Che91, CD89].

This becomes a problem if the locking semantics desired by the higher level and the locking semantics provided by DSM are different. If locking is integrated with coherence in DSM, implementing a change in the desired locking semantics can require a change in the DSM protocol.

For example, suppose a transaction system desired to implement a form of *conflict-based* locking [Wei84] instead of read/write locking. If two transactions attempted to write to the same page but the transactions commute and hence do not conflict, they should be able to modify the same page simultaneously.[4] In order to support this, Ramachandran-Khalidi DSM would have to be modified to support different locking logic, and detect when read/write locking logic should be used and when conflict-based locking logic should be used. Currently, Ramachandran-Khalidi DSM is incapable of making such distinctions. It only supports standard read/write locking.

Moreover, in the above case, if locks on the same page were granted to two transactions, the DSM system would have to maintain coherence on the page while two (or more) transactions simultaneously work with the page. Maintaining such coherence would require modifying the Ramachandran-Khalidi DSM protocol since the existing protocol assumes that only one site can gain a lock on a page held in read/write-mode at one time.

Since locking can be kept out of DSM by using a distributed lock manager, we felt that locking *can be and should be kept out of DSM.* Our implementation of IBCC uses a distributed lock manager that is orthogonal to DSM. Locking requests are sent to the lock manager and data requests are sent to DSM. The lock manager handles locking requests issued by the kernel and the DSM mechanism handles fetching the page and maintaining coherence.

Distributed lock management can be expensive because of the overhead involved in inter-site communication.[5] Orthogonal lock management and DSM service results in generating messages for both locking and coherence whereas integrating DSM with locking combines the locking messages with the coherence messages. Our work-around for that problem was to use logical locks. One lock controls a set of one or more pages (possibly non-contiguous) that we call a *locking segment.* Thus, locking a locking segment generates a small fixed number of messages (a request and result per lock request) since the entire locking segment is locked/unlocked at once. Given that in our implementation, transactions use logical locks and thus lock an entire set of pages with one lock, we feel that the flexibility of an orthogonal lock manager is worth the cost of performance. Moreover, using an orthogonal lock manager makes it much easier to change the implementation should the desired locking semantics ever change. Since we depend on DSM only for coherence, the locking semantics can be changed at will without affecting the DSM coherence system, protocol, and/or implementation.

Thus locking turned out to be orthogonal to DSM. However, we were hoping that we would be able to use the locking mechanism already present in Ramachandran-Khalidi DSM. Instead we were forced to implement our own orthogonal locking manager.

5 Version Management and DSM

The atomicity property of transactions is due to two features: locking and version management. Locking preserves view atomicity. Version management, which includes version creation, version preservation and commit/abort processing, provides failure atomicity. When

[4]To maintain failure atomicity, commit/abort handling would probably involve user-defined handlers.

[5]However, there is significant research potential in this area (migratable locks, hierarchical locks, etc.).

we first looked at integrating DSM and IBCC, we were hoping that DSM and transaction-based version management were orthogonal and that IBCC version management could be layered on top of a DSM store as if the DSM store were a local storage device. Unfortunately, we were wrong.

Consistency control mechanisms must ensure that the state of persistent data is preserved before allowing a consistent computation to alter the data. This preservation is necessary to ensure that the original state of the persistent data can be restored should the computation abort. Transaction mechanisms usually save the state by creating an intermediate or shadow copy of the data about to be altered. The transactions work on the newly-created intermediate copies. The transaction mechanism will atomically replace the persistent versions with the intermediate versions during commit processing, thus atomically updating the persistent data.

Consistency mechanisms must therefore be able to determine when a version must be created, and then must correctly preserve the older version. In a distributed non-DSM environment, since the locality assumption holds, system designers can assume first, that all computations that attempt to alter a version reside on the local site, and second, that the version being referenced itself resides on the local site. Given these two assumptions, designers can ensure that all the information necessary to determine whether a version has already been preserved also resides on the local site. Thus, in a non-DSM environment, intercepting an attempted update of a version, deciding if the version must be preserved before allowing the update to proceed, and actually preserving the version all involves only one site: the site where the persistent data resides.

Moreover, the version preservation mechanism can be easily implemented as an orthogonal mechanism layered on top of the backing storage mechanism. Intermediate versions can be created in volatile storage. Commits can be implemented by first writing the intermediate versions to a stable log before writing them into the backing store.

We thought that the same should hold true in a DSM environment. Ideally, the DSM mechanism should be unaware of the versions of data being used by a transaction just like the file system is unaware of the versions in centralized transaction systems. The obvious way to implement version creation on top of DSM is to use the following strategy:

- When a transaction touches a data item, it obtains a lock on the data from the lock manager.
- After successfully locking the data, the transaction gets the data from the DSM server.
- Then the transaction copies the pages it has obtained into a different set of locally allocated pages (temporary version) and use these pages instead of the actual data.
- After the transaction commits, it copies the local versions into the pages it obtained from the DSM server and returns them to the server, thus updating the DSM version. It would then unlock the data. If the transaction aborts, the local version is simply discarded.

However, reality ruled out this intuitively obvious approach.

5.1 Version Management

The problem is that correct version management in a DSM-environment may involve more than one site. In addition interactions between transactions and processes (allowed by

IBCC) make version management even more complicated.

Consider the following situation: a transaction T on site A performs an invocation on object O and alters a page of the data as part of that invocation. A new version of that page is created locally. Now, if T accesses that page again, before committing, T should see the new version of that page, regardless of where T is currently executing.

In a non-DSM environment, ensuring that T always sees the latest version of the page is relatively straight-forward since an invocation of object O by transaction T will always run on the same site unless O is migrated by the operating system. However, if O is migrated, all the information associated with O can be moved as well.

In a DSM-environment, however, multiple invocations of object O by transaction T may not run on the same site. For load-balancing reasons, the system may choose to run additional invocations of object O by transaction T on different sites. Therefore, before creating a new version of a page, the consistency control mechanisms on a site must first determine whether a new version of that page has already been created *anywhere* in the distributed system. If this determination is not made or the information is incorrect, the version will be handled incorrectly and consistency will fail. Therefore, up-to-date, correct version information (whether a version of a page has already been created and if so, how many) must be available to all sites.

We partially addressed the version information management problem by implementing a Transformation Information System (TIS) that maintains version creation information. We associated a *t-segment* with each transaction that contains information on what versions have been created by that transaction. When a transaction accesses data, the t-segment can be consulted to see if a new version of that data should first be created. In this case, we were able to make DSM work for us by implementing the t-segment as a relocatable data structure that is DSM-controlled.

Note that the TIS system manages version information, it does not address the version creation and preservation problems. These are discussed in the next section.

We were able to implement the TIS orthogonally from DSM because of *one* reason: transactions in our system are single-threaded. While the version creation information for each transaction must be available to every site, the information will be used at only site at a time since a transaction can be active at only site at a time. Only one object invocation will be running. The remainder of the invocations will be suspended, waiting on the termination (return) of other invocations. If we had supported multi-threaded transactions, this approach would have resulted in terrible performance. The t-segment is frequently used when processing page-faults by a transaction and would therefore be a contention hot-spot.

5.2 Version Creation/Preservation

Versions creation and preservation are really duals. Before creating a new version, the state of the current version must first be preserved so that it may be restored in case of an abort. The pages of the new version must be initialized to the state of the old pages at the time of creation.

When we examined the problem of version preservation in a DSM environment, there appeared to be two alternatives. First, we could either implement a version preservation mechanism independently of DSM such as a mechanism to preserve versions locally (either by using local disks or by using a second DSM-controlled store), or second, we could modify DSM to support version creation.

We chose to modify DSM to support version creation, and we did so for two reasons.

First, *Clouds v.2* supports both processes and transactions (or transaction equivalents to be precise). That means that when a transaction modifies data and then aborts, the original state of the data should be restored, even if that state was produced by a process and therefore never explicitly committed to stable storage by the process.

Second, transactions in our scheme can modify segments that consist of one or more pages. Thus, version preservation applies to entire segments, not just individual pages. When a segment is preserved, semantically, *all* the pages of the segment must be preserved.

Supporting these semantics implies that the entire state of the segment *must* be preserved when a transaction updates a segment that has been previously written to by a process. In other words, a new version consisting of *all* the pages of the old version must be created (and initialized) and no page of the old version can be used by the transaction. This is necessary to ensure that the segment can be rolled back to the state it was in immediately prior to the moment it was first touched by the transaction. Even if processes and transactions are not allowed to access the same data concurrently, this version creation/preservation problem exists in DSM systems.

Version preservation can be done in two ways: in a "brute-force" fashion by preserving (copying) all pages of the segment, or in a "lazy" fashion (on demand) by preserving only those pages that are currently in use and only preserving the other pages just before they are requested for use. The lazy method is analogous to using a copy-on-write technique. If segments can consist of a large number of pages (such as in a database, for example), the brute-force method is clearly infeasible as it requires copying the entire segment.

Implementing the lazy method requires help from the DSM server. This technique requires knowledge of which pages are currently being used at the time the segment is preserved. The pages in use *must* be preserved before the transaction is allowed to work on the segment since the system must be able to restore the state of the segment back to the state that existed *immediately* before the transaction first modified the segment (triggering the version preservation). Thus, in order to preserve the segment efficiently, someone has to contact all sites that are working with the pages of the segment about to be preserved — in other words, all keepers of pages in the segment. However, *only* the DSM server that "owns" the segment being preserved knows which pages are being used and which sites are keepers of those pages. In order to implement version preservation orthogonally of DSM, we would either have had to move all the pages to the site creating the version (brute force) or send broadcast messages to all sites to determine if anyone was using any part of the segment (unnecessarily gathering information already available to DSM). Therefore, we decided to augment DSM to perform version preservation.

We added a *version_create(segment)* operation. The *version_create(segment)* operation causes the DSM owner site to yank all outstanding pages back to the owner site and record the fact that a new version of that segment has been created. After the version creation is recorded, *gets* semantically fetch pages of the new version. *discards* of those pages do not overwrite pages in the preserved version but instead are stored separately so that the segment can be rolled back to its prior state.

6 Commit/Abort Processing

Commit processing in a distributed, non-DSM environment typically involves executing some form of a two-phase commit. The committing site acts as the coordinator and directs all sites where the transaction has modified persistent data to pre-commit the changes and then return a status code to the coordinator. If all sites successfully pre-committed, the coordinator logs the commit and informs all participating sites that the commit has succeeded. Otherwise, the coordinator informs all participating sites that the commit failed. No other transaction may access the data being committed until after the commit terminates.

6.1 Committing and Aborting with DSM

Committing on a DSM system cannot be done similarly.[6] The transaction may have visited several sites and may have modified data while running on all those sites. Some of the same data may also have been modified on different sites. Only the DSM mechanism truly knows where all the data resides when the commit begins. Thus, at a minimum, we had to involve the DSM server in retrieving the latest versions of the data that have been updated.

Moreover, transactions usually commit multiple pages and these pages *must* be committed or aborted *atomically.* This implies that either all the pages have to be simultaneously committed wherever they happen to be located on the system, or that all the pages must be brought to one site and atomically committed from or at that site. Under Li-Hudak DSM, it is simply impossible to gather all the pages to one site and hold them there for an indefinite period of time (necessary to complete the commit) without modifying the DSM protocol and/or algorithms. With Li-Hudak DSM, at any point in time, a page may be yanked away from the keeper to be read or modified on some other site. An atomic commit is clearly impossible under these circumstances.

With Ramachandran-Khalidi DSM and Fleisch DSM, it is possible to lock or pin the pages at a site. However, since DSM must already keep track of versions, a commit cannot be performed without directing DSM to destroy one of the versions. Therefore, the DSM system must be modified so that the owner (i.e. the DSM server) can be notified of the commit. In addition, since the DSM owner holds the persistent (stable) versions of the segment, all modified pages must be transmitted back to the owner anyway to be written out to stable storage. So it makes sense, even if pages can be pinned or locked to a site, to further modify DSM to actually perform the commit, since the owner must write the pages out to storage and the owner can most efficiently retrieve the latest versions of all outstanding pages in its segment.

6.2 DSM Commit Implementation

When augmenting DSM to support commit/abort processing, we decided to maintain the invariant that the owner site should be the only site to write segment pages to the disk (regardless of whether the pages are intermediate or persistent versions). To handle the commit operation, we added *pre-commit* and *commit* operations for use in the two-phase commit process, and *pre-commit*, *commit*, and *yank* messages to the DSM protocol. The *commit(A)* operation directs the DSM mechanism to commit segment A. The commit is synchronous. The commit is assumed to have succeeded (or failed) once the operation returns with a success (or failure) code.

The pre-commit operation causes a *pre-commit* message to be sent to the owner site for the data. The owner site then executes the pre-commit. Since the data is stored at the owner site in our DSM implementation, we modified the owner to issue *yank* messages to all keeper sites that are updating (writing) the pages they hold. The *yank* message directs the keeper to send a copy of the page to the owner, thus enabling the owner to save/preserve

[6]But you expected that by now, right?

the current version of the page. Note that a *yank* message does not alter the keeper of the page and therefore does not imply an *invalidate* or *discard.* The *yank* message merely requests the latest state of the page.

Yanking back outstanding pages may be necessary in systems supporting nested transactions since a nested transaction may modify data that its parent has also modified but not yet committed, creating multiple versions of the same data. The changes made by the parent must thus be preserved before allowing the child to proceed in case the child aborts.

In addition, if the system supports both processes and nested transactions, processes on the current keeper site may have made changes to the data that have not been written back to the owner site. Therefore, the page must be yanked back to the owner to save those changes before allowing a transaction to work on the data.

6.3 DSM Abort Implementation

Abort processing in a distributed, non-DSM environment is fairly straightforward. The consistency control mechanism simply directs all sites where the transaction has created new intermediate versions to throw away all the intermediate versions. Future references to the affected data will reference the original unmodified versions.

As in the case of commit processing, once DSM is added, pages containing intermediate versions may be paged to any site on the network. To abort a transaction correctly, all intermediate versions must be thrown away regardless of where they are located on the network.

We therefore added an *abort(segment)* operation to the DSM design that directs the DSM mechanism to abort the intermediate versions of data created by the transaction. In our current implementation, the DSM mechanism sends the abort request to the owner site who then executes the abort processing.

The abort processing consists of sending *invalidate* messages to the keeper sites of all pages that have had intermediate versions created. The latest versions must be thrown away since they contain changes that are to be aborted. Further references by threads to the data will result in the sites asking the owner site for another copy of the page(s). The owner can then send them the correct (older) version. Note that unlike the *yank* message used in pre-commit processing, abort processing invalidates all outstanding modified pages.

6.4 Single-object Commits with DSM

One interesting side-effect of integrating DSM with consistency control is the effects of the integration on single-object commits. IBCC supports the notion of single-object (locally consistent) transactions. In a non-DSM environment, these transactions would be fast. All data access is local, all locking is local, and the commit is purely local, thus a multi-site 2-phase commit is not necessary.

However, once DSM is added in, all of those guarantees disappear. Data access may be remote since the object may be in use at multiple sites simultaneously. Locking may remote if the object and/or lock does not currently reside at the local site. Finally, the commit automatically involves multiple sites since the keeper and owner are involved and may involve multiple keeper sites if the object data or computation have migrated because of DSM.

7 Conclusion

The rationale behind the design and implementation of IBCC was to build a consistency control mechanism that is more flexible than standard transactions. A part of the IBCC implementation works like transactions. Since implementing transactions are well understood, we assumed that we would have very little trouble in implementing the transactional behavior and would spend our time working out the problems with other features of IBCC.

However, implementing the transaction-like features on top of DSM turned out to be quite a job. The version management and commit processing required major surgery on the existing DSM subsystem. We were forced to add version and commit/abort management, the details of which are intricate and somewhat non-intuitive. In addition, in the instance where we were hoping to use an existing DSM locking capability, we were forced instead to use a separate, orthogonal distributed locking manager (implemented by R. Ananthanaryan). We did however finish the IBCC implementation and that effort is reported in [CD91].

DSM presents an appealing simplifying illusion: everything is local. However, while this illusion simplifies the life of distributed system users, we found that this illusion can introduce tremendous complications for implementors of distributed operating systems.

References

[CD89] Raymond C. Chen and Partha Dasgupta. Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation. In *Proceedings of the 9th IEEE International Conference on Distributed Computing Systems (IC-DCS)*, June 1989.

[CD91] Raymond C. Chen and Partha Dasgupta. Implementing Consistency Control Mechanisms in the *Clouds* Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[Che91] Raymond C. Chen. *Consistency Control and Memory Semantics for Persistent Objects.* PhD thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA, 1991.

[DCM$^+$90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the *Clouds* Distributed Operating System. *Computing Systems*, 3(1), 1990.

[DRA91] Richard J. LeBlanc Dasgupta, Partha, Umakishore Ramachandran, and Mustaque Ahamad. The *Clouds* Distributed Operating System. *IEE Computer*, 24(11), November 1991.

[FP89] Brett Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, Digital Equipment Corporation 1989.

[Kah89] M. Yousef A. Kahlidi. *Hardware Support for Distributed Object-based Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, June 1989.

[LH86] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. 5th ACM Symp. Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.

[PD88] David V. Pitts and Partha Dasgupta. Object Memory and Storage Management in the *Clouds* Kernel. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Eighteenth Annual International Conference on Parallel Processing*, August 1989.

[Spa86] Eugene H. Spafford. *Kernel Structure for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[Wei84] W. E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1984.

TRANSPARENT FAULT-TOLERANCE IN PARALLEL ORCA PROGRAMS

M. Frans Kaashoek (kaashoek@cs.vu.nl)
Raymond Michiels (raymond@cs.vu.nl)
Henri E. Bal (bal@cs.vu.nl)
Andrew S. Tanenbaum (ast@cs.vu.nl)

Vrije Universiteit, Amsterdam
The Netherlands

ABSTRACT

With the advent of large-scale parallel computing systems, making parallel programs fault-tolerant becomes an important problem, because the probability of a failure increases with the number of processors. In this paper, we describe a very simple scheme for rendering a class of parallel Orca programs fault-tolerant. Also, we discuss our experience with implementing this scheme on Amoeba.

Our approach works for parallel applications that are not interactive. The approach is based on making a globally consistent checkpoint from time to time and rolling back to the last checkpoint when a processor fails. Making a consistent global checkpoint is easy in Orca, because its implementation is based on reliable broadcast. The advantages of our approach are its simplicity, ease of implementation, low overhead, and transparency to the Orca programmer.

1. INTRODUCTION

Designers of parallel languages frequently ignore fault tolerance. If one of the processors on which a parallel program runs crashes, the entire program fails and must be run again. For a small-scale parallel system with few processors, this may be acceptable, since processor crashes are unlikely. The execution-time overhead of fault tolerance and its implementation costs may not be worth the extra convenience. After all, the goal of parallelizing a program is to reduce the execution time.

Consider, however, a parallel program that runs on a thousand CPUs for 12 hours to predict tomorrow's weather. If the mean-time-between-failure of a CPU is a few years, the chances of one of them crashing can no longer be neglected. Moreover, if a processor crashes when the computation is almost finished, the whole program has to be started all over again, thus doubling its execution time. In general, the larger the scale of the parallel system, the more important fault tolerance becomes. Future large-scale parallel systems will have to take failures into account.

An obvious question is: Who will deal with processor crashes? There are two options. One is to have the programmer deal with them explicitly. Unfortunately, parallel programming is hard enough as it is, and fault tolerance will certainly add even more complexity. The alternative is to let the system (i.e., compiler, language run time system, and operating system) make programs fault-tolerant automatically, in a way transparent to programmers. The latter option is clearly preferable, but, in general, it is also hard to implement efficiently.

In this paper, we will discuss how transparent fault tolerance has been implemented in the Orca parallel language [Bal 1990]. Orca is a language for running parallel programs on distributed systems, such as the Amoeba system [Tanenbaum et al. 1990]. The failures that we consider are transient failures such as hardware errors.

The problem we have tried to solve is modest: to avoid having to restart long-running parallel Orca programs from scratch after each crash. The goal is to do this without bothering the programmer and without incurring significant overhead. We have not tried to solve the more general class of problems of making all distributed systems fault-tolerant. Instead, we consider only the class of parallel programs that take some input, compute for a long time, and then yield a result. Such programs, for example, do not interact with users or update files.

Our solution is simple: make global checkpoints of the global state of the parallel program using reliable broadcasting. If one of the processors crashes, the whole parallel program is restarted from the checkpoint rather than from the beginning. Users can specify the frequency of the checkpoints, but otherwise are relieved from any details in making their programs fault-tolerant.

The key issue is how to make a global checkpoint that is *consistent*, preferably without temporarily halting the entire program. It turns out that this problem can be solved in a surprisingly simple way in Orca.

The outline of the rest of the paper is as follows. We first give some information about the Orca language and its implementation on Amoeba. Next, Section 3 describes the design of a fault-tolerant Orca run time system. Section 4 gives the implementation details and the problems we encountered with Amoeba. In Section 5, we give some initial performance measurements. In particular, we show how much time it takes to make a checkpoint and to restart a program after a crash. In addition, we measure the overhead of checkpoints on example Orca programs. Section 6 compares our method with related approaches, such as explicit fault-tolerant parallel programming, message logging, and others. Finally, in Section 7, we present some conclusions and see how our work can be applied to other systems.

2. THE ORCA LANGUAGE AND ITS IMPLEMENTATION

In this section we give a brief description of the Orca language and its implementation. The goal is just to give enough detail to make the rest of the paper understandable. More detailed descriptions are given elsewhere [Bal 1990; Bal et al. 1990, 1992].

2.1. Orca

Orca is a language for running parallel programs on distributed systems. Although Orca is intended for systems without physical shared memory, its programming model is based on shared data. Processes in Orca communicate through *shared data-objects*, which are variables of abstract data types. Processes can share objects even if they run on different machines. The objects are accessed solely through the operations defined by the abstract data type.

Initially, an Orca program consists of a single process, but new processes can be created explicitly through a *fork* statement. The parent process can pass any of its data-objects as a shared parameter to the child. In this case, the data-object will be shared between the parent and the child. The parent and child can communicate through this shared object, by executing the operations defined by the object's type.

The semantics of the model are very simple. All operations are applied to single objects, and all operations are executed indivisibly. The latter property simplifies programming, since users do not have to worry about what happens if two operations are applied simultaneously to the same object. In other words, mutual exclusion synchronization is done automatically.

The first property is a restriction, since it rules out atomic operations on collections of objects. This restriction, however, makes the model efficient to implement, because no complicated two-phase update protocols are needed. As it turns out, parallel applications seldom need atomic operations on multiple objects [Bal 1990]. If needed, however, they can be constructed in Orca, by building them as sequences of simple operations. In this case, the programmer must explicitly deal with synchronization.

Orca is perhaps best thought of as a programming language approach to Distributed Shared Memory (DSM). Other systems (e.g., IVY [Li and Hudak 1989]) try to simulate physical shared memory on a distributed system and provide the same operations (read/write words) as real memory. Orca provides a DSM programming model, but the operations on the shared memory are defined by the programmer through abstract data types. Also note that Orca is *not* an object-oriented language; it is a procedural language with abstract data types. Unlike in concurrent object-oriented languages, objects in Orca are purely passive (hence the name data-objects). Furthermore Orca does not support inheritance.

2.2. A distributed implementation of Orca

Orca can be implemented efficiently on a distributed system using a run time system (RTS) that replicates shared objects in the local memories of the processors [Bal et al. 1989]. If a processor has a local copy of an object, it can do *read-operations* locally, without doing any communication. A read-operation is an operation that does not modify the object's local data; read-operations are distinguished from write-operations by the Orca compiler.

After a write-operation, the copies of the object will no longer be up-to-date. There are many ways of dealing with this situation. In the implementation described here, write-operations are broadcast to all nodes containing a copy. All these nodes update their copy by applying the write-operation to the copy.

A key problem is how to update all the copies of an object in a consistent way. We solve this problem using an *indivisible* reliable broadcast protocol. With such a protocol, all messages are delivered reliably at all receivers. In addition, the protocol guarantees that, if two processes P1 and P2 simultaneously try to broadcast two messages (say M1 and M2), then either all receivers get M1 first or all receivers get M2 first. Because the broadcast is indivisible, it is not possible that some processes will see M1 first while others get M2 first. Our protocol in fact assigns consecutive *sequence numbers* to the broadcast messages; each receiver handles the incoming messages exactly in the order of their sequence numbers. Due to space limit the protocol, its implementation, and its performance cannot be discussed in detail here, but readers are referred to [Kaashoek and Tanenbaum 1991; Tanenbaum et al. 1992].

2.3. Running Orca programs on Amoeba

The Orca implementation runs on the Amoeba distributed operating system [Tanenbaum et al. 1990; Mullender et al. 1990]. Amoeba is based on the processor pool model. Each user has his or her own workstation, but the real computing power is located in a pool of processors shared among all users. These processors are connected through a LAN and are allocated on demand. Processes can communicate with each other using at-most-once Remote Procedure Call (RPC) [Birrell and Nelson 1984] or using group communication [Kaashoek and Tanenbaum 1991]. The group communication primitives implement the broadcast protocol mentioned above.

Parallel Orca programs are run on the processor pool. An Orca application is started by the program *gax* (Group Amoeba eXecute). *Gax* asks the directory server for the capability for the program, and with the capability it fetches the Orca executable (Orca program linked with the Orca RTS) from the file server. Next, *gax* allocates the requested number of processors in the processor pool and starts the executable program (see Fig 1) on each processor. These Orca processes together form one process group. A message sent to the group is received in the same order by all processes (including the sending process), even in the presence of communication failures.

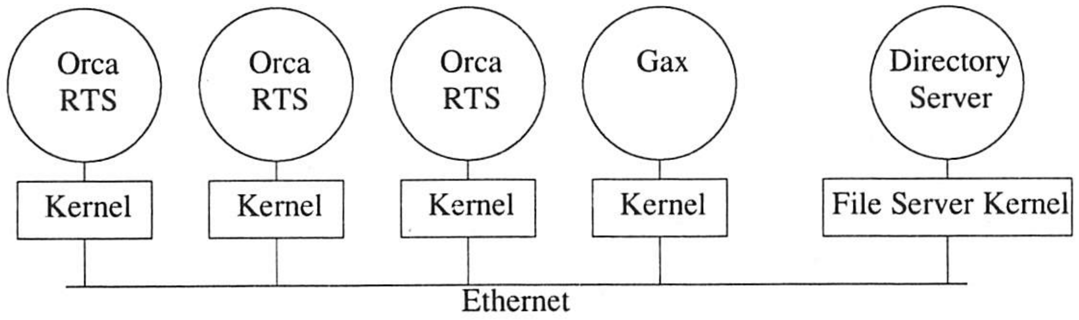


Fig. 1. An Amoeba system with an Orca application running on 3 processors. The Orca processes together form one process group.

3. DESIGN OF A FAULT-TOLERANT ORCA RUN TIME SYSTEM

There are many ways of making Orca programs fault-tolerant. One approach is to ask a special server to keep an eye on Orca applications and start them again if they fail. (In Amoeba such a server is available and is called the *boot server*). Unfortunately, this does not win much, since the application would have to start all over. For applications that have a deadline this is not appropriate.

Another approach is to use a message logging and playback scheme, such as optimistic recovery [Strom and Yemini 1985; Johnson 1989]. Message logging is designed for general distributed applications rather than just parallel applications, and may be too expensive for parallel applications. Optimistic recovery, for example, can deal with interactive programs. Programs using optimistic recovery will not ask for the same input twice or produce the same output twice. While this property is useful, it is not essential for most parallel programs, which frequently are not interactive. For those programs, the message logging solution is overkill. A cheaper and simpler form of fault tolerance is desired.

The method we use is to periodically make a global checkpoint of all the Orca processes. After a crash, all processes continue from the last checkpoint. Since parts of the program may be executed multiple times, the method cannot be used for interactive programs.

The most important design issue is how to obtain a *consistent* global checkpoint. As an example, assume a process P1 makes a checkpoint at time T1 and then sends a message to process P2. (Although Orca programs do not send messages, their run time systems do.) Assume that process P2 receives this message and then makes a checkpoint, at time T2. Obviously, the two checkpoints are not consistent. If both processes are set back to their state of their checkpoint, P1 would again send the message, but P2 would be in a state where it had already accepted the message. So, P2 would receive the message twice.

As explained in [Koo and Toueg 1987], checkpointing should be done consistently relative to communication. It certainly is not necessary to have all processors make checkpoints at exactly the same time, but messages must not cross checkpoints.

Based on this observation, one could envision making a consistent checkpoint by first telling each processor to stop sending messages. When the whole system is quiet, each processor is instructed to make its local checkpoint; all these checkpoints will then be consistent, since no interprocess communication takes place during checkpointing. We will not go into the details of how such a design might work. Suffice it to say that this solution would not be very attractive. The reason is that it may take a lot of time and overhead to bring the whole (distributed) system to a halt. Also, the more processors there are, the more time is wasted.

In the Orca run time system based on reliable broadcasting it is almost trivial to make consistent checkpoints. As we explained in Section 2, the Orca run time systems communicate through reliable indivisible broadcasting. All processes receive all broadcast messages in the same order. Therefore, to make the global checkpoint consistent, all that is needed is to broadcast one *make-checkpoint* message. This message will be inserted somewhere in the global order of broadcast messages. Assume that the *make-checkpoint* message gets sequence number N in this global ordering. Then, at the time a process makes its local checkpoint, it will have received and handled messages 1 to $N-1$, but not messages $N+1$ and higher. This applies to *all* processes. Therefore the checkpoints are all consistent.

To recover from a processor crash all processes synchronously roll back to their last checkpoint. No process will send a message before all processes have been rolled back. Thus, when a process sends a message, all processes will receive it with the same sequence number. Because all processes roll back and synchronize before they continue running, they will start in a consistent state.

For some applications checkpoints clearly are not going to be cheap. Each process must save its local data (or, at the very least, the difference with the previous checkpoint). For some processes, this may easily involve writing several hundred kilobytes to a remote file server. Even with the Amoeba Bullet file server [Van Renesse et al. 1989], this may take a substantial fraction of a second.

With multiple processes, things will get worse. The time to make a global checkpoint will depend on the configuration of the distributed system and on the network. Clearly, having a single centralized Bullet server for a thousand pool processors will not be a good idea. However, the Bullet server can be (and is) replicated, so different pool processors can use different instantiations of this server.

The main advantage of our algorithm is that it is extremely simple and easy to implement. It is not an optimal algorithm, but it is still useful. In particular, for long-running parallel applications, the overhead of making a checkpoint every few minutes will be quite acceptable. An advantage of our scheme is that the frequency of the checkpointing can easily be changed. Using a lower frequency will decrease the overhead, but increases the average amount of re-execution due to crashes. We will get back to this performance issue in Section 5, where we give initial performance results.

4. IMPLEMENTATION ON AMOEBA

In this section, we will describe the problems encountered with implementing a fault-tolerant run time system for Orca on Amoeba. To understand the implementation we have to take a closer look at how process management is done in Amoeba. Each Amoeba kernel contains a simple process server. When *gax* starts an Orca application on a processor, it sends to the processor's process server a descriptor containing capabilities for the text, data, and stack segment. The process server fetches the segments from the file server, builds a process from the segments, and starts the process. The capability for the new process, called the *owner capability*, is returned to *gax*.

To checkpoint a single process, *gax* sends a *stun* signal to the process server that manages the process. The process server stops the signaled process when it is in a safe state, that is, when it is not in the middle of an RPC. (Interrupting a process while doing a RPC would break the at-most-once semantics.) When the process has stopped, the process server sends the process descriptor to the owner (*gax* in this case) and asks it what do to with the process. Using the capabilities in the processor descriptor, *gax* can copy the process state to the file server. After having copied the state, *gax* tells the process server to resume the process.

Making a global checkpoint of the complete Orca application now works in the following way. Every *s* seconds, a thread in the RTS of one of the machines broadcasts a *make-checkpoint* message to all other processes in the application. When a process receives this globally ordered message, it asks *gax* to make a checkpoint of it, as described above. When all processes of the application have made a checkpoint, *gax* stores the capabilities for the checkpoints with the directory server. Other Amoeba servers will make replicas on multiple file servers using the capabilities stored with the directory server.

Rolling back to a previous checkpoint works as follows. If a member of the group that runs the Orca application crashes, the group communication primitives return an error after some period of time. When a process sees such an error, it asks *gax* to roll the application back. *Gax* starts by killing any surviving members and then starts the application again. Instead of using the original executable, it uses the checkpoints of the processes.

The actual implementation is more complicated due to a number of problems. The first problem is that by using *gax* we have introduced a single point of failure: if *gax* crashes, the Orca application cannot make any checkpoints or recover. To prevent this from happening, *gax* is registered with the boot service. The boot service checks at regular intervals whether *gax* is still there. If it is not there, the boot service starts *gax* over. (When *gax* starts running again, it kills the remaining processes and rolls the application back to the last checkpoint.) The boot service itself consists of multiple servers that check each other. As long as the number of failures at any point in time is smaller than the number of boot servers, the Orca application will continue running.

A second problem is that the checkpoints made by the process server do not contain all the state information about the parallel program. In particular, the kernel state information about process groups is not saved.

As an example, suppose the RTS initiates a global checkpoint by broadcasting a *make-checkpoint* message. At about the same time, a user process executes a write-operation on a shared object. As a result, its local RTS will send an *update* broadcast message and block the Orca process until this message has been handled. Assume that the broadcast protocol orders the *update* message just after the *make-checkpoint* message. The broadcast protocol will buffer this message in the kernel and it will be delivered after the checkpoint is made. If after a crash a process has been rolled back to this checkpoint, all the kernel information about the group is gone, including the buffered message.

Fortunately, detecting that a message has been sent and not received by any process when the checkpoint was made is easy. After a thread has sent a broadcast message, it is blocked until the message is received and processed by another thread in the same process. If after recovery there are any threads blocked waiting for such events, they are unblocked and send the message again.

The problem that the kernel information about the group is not included in a checkpoint is harder. We have solved this problem by having *gax* maintain a state file, in which it keeps track of additional status information. This file contains the current members of the process group, as well as the port names to which the restart messages (discussed below) are to be sent, the number of checkpoints made so far, and other miscellaneous information.

As a consequence of this approach, *gax* must read the status file during recovery and rebuild the process group. To rebuild the group, *gax* needs the help of the processes that are being rolled back. These processes must actively join the newly formed process group. Clearly, all this activity is only needed during recovery and not after making a checkpoint. The problem is, it is difficult for the processes to distinguish between these two cases (i.e., resuming after making a checkpoint and resuming after recovery). After all, the state of the parallel program after recovery is the same as the state of the latest checkpoint.

Our solution to this problem is as follows. After making a checkpoint, a checkpoint server does not continue the process immediately, but it first waits for a *continue* message from *gax*. If it receives this message, it simply continues. On the other hand, if a processor has crashed and the program has just recovered, *gax* sends a *restart* message instead of the usual *continue*. If the checkpoint server receives a *restart*, it first participates in rebuilding the group state, before continuing the application.

Yet another performance issue concerns the text segment of a process. It is not necessary to dump the text (code) segment of each process, since text segments do not change and can be obtained by reading the executable file containing the process's image. At the expense of writing some more code, our prototype implementation avoids saving the text segment of a process after the first checkpoint.

Another important issue is the scalability of our implementation. The cost of broadcasting the *make-checkpoint* message is almost independent of the number of receivers [Tanenbaum et al. 1992], and therefore scales well. However, *gax* and the bullet server are likely to become a bottleneck as the number of processors increases. This could be avoided by using a distributed algorithm for making checkpoints, for example, by sending the checkpoints to a neighbour instead of to the bullet service.

Although the implementation is more complicated than we expected, only minor modifications were required to existing software. We added 324 lines of C-code (including 84 lines of comments) to the Orca RTS, bringing the total number of lines of C-code for the RTS at 6196. To the sources of *gax* we added 779 lines. No changes were made to the Amoeba kernel.

5. PERFORMANCE

We have measured the performance of the implementation described in the previous section. The Orca programs run on a collection of MC68030s. The directory server and the file server are duplicated. They run on Sun 3/60 and use a SCSI-3 controller and a WREN IV SCSI disk. All processors are connected by a 10 Mbit/s Ethernet. Given this environment, an Orca program running on n pool processors can tolerate 1 failure of the directory server or file server and $n-1$ concurrent processor pool failures. If both directory or both file servers crash, the Orca program will block until one of each is back up. If n concurrent failures happen, the Orca program will be restarted from the latest checkpoint by the boot server. Because all processors are connected by one network, the system will not operate if the network fails. If the system would have contained multiple networks we could have tolerated network failures, because the Amoeba routing protocol is based on dynamic routing tables [Kaashoek et al. 1991].

We have measured the overhead of checkpointing and recovery for a toy Orca program called *pingpong*. Pingpong consists of processes running on different processors and sharing one integer object. Each process in turn increments the shared integer and blocks until it is its turn to increment it again. This program is interesting because it sends a large number of messages: for each increment of the shared integer the RTS sends one message. If this program were to run on a system based on message logging, it would experience a large overhead.

We ran pingpong with and without checkpointing and computed the time for making one checkpoint. The results are given in Fig. 2. Each process has 7 segments: one text segment of 91 Kbytes, one data segment of 56 Kbytes, and 5 stack segments of 11 Kbytes each. The state file is 3670 bytes large. The first checkpoint with text segment consists of 202 Kbytes and subsequent checkpoints (without text segment) are 111 Kbytes. If pingpong is running on 10 processors, taking a global checkpoint will involve writing 1.11 Mbytes to the file server. With 8 or more processors, the bullet service becomes a bottleneck, because it is only duplicated.

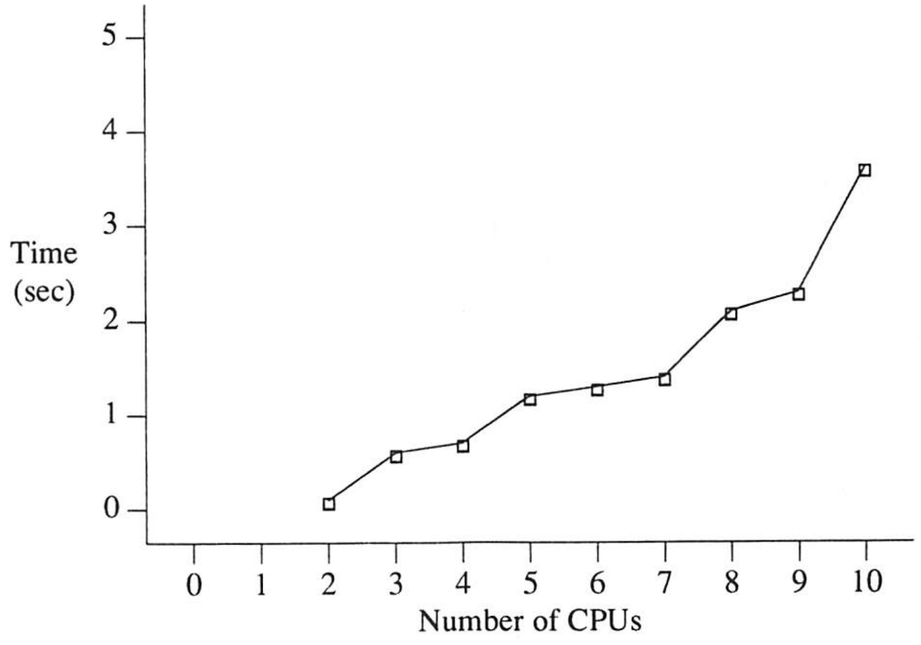


Fig. 2. The cost of making a checkpoint.

The time to recover from a processor crash is equal to the time to detect the processor crash plus the time to start *n* processes. The time to detect a processor crash is tunable by the user. The user can specify how often the other members in the group should be checked. The time to start a new process on Amoeba is 58 msec [Douglis et al. 1992]. Thus, most of the time for making a global checkpoint is spent in transferring segments from each processor to the file server.

For any application the total overhead (the difference between execution time with and without checkpointing) introduced by our scheme depends on 3 factors:

1) The cost of making a checkpoint (dependent on the number of processors and the size of the process's image);

2) The cost of a roll back;

3) The mean time to failure (MTTF) of the system (hardware and software).

If one wants to minimize the average run time of the application, then given the numbers for these factors, one can compute the optimal computing interval (the time between two checkpoints such that the average run time is minimized) and thus the overhead introduced by checkpointing (see Appendix):

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{(T_{comp} + T_{cp}) \,/\, 2 + T_{rb}}{T_{MTTF}},$$

where T_{comp} is the computing interval, T_{cp} is the mean time to make a checkpoint, T_{rb} is the mean time to recover from a crash.

If, for example, an application runs for 24 hours using 32 processors, the cost for a checkpoint is 15 seconds, time to roll back is 115 seconds, and the MTTF is 24 hours, then the optimal checkpoint interval is 27 minutes. In this case the overhead is 1.6 percent. If the MTTF is 7 days, then the optimal checkpoint interval is 71 minutes and the overhead is 0.7 percent.

6. COMPARISON

Although considerable research has been done both on parallel programming and on fault tolerance, few researchers have looked at fault-tolerant parallel programming. In this section, we will look at some of this work and also at more general techniques for fault tolerance.

6.1. Fault-tolerant parallel programming

An alternative to our approach is to let programmers deal with processor crashes. We have described our own experiences with explicit fault-tolerant parallel programming in a separate paper [Bal 1992]. The language used for these experiments was Argus [Liskov 1988]. Below we will first compare our work on Argus and Orca.

The Argus model is based on guardians, remote procedure calls (RPC), atomic transactions, and stable storage. A guardian is a collection of processes and data located on the same processor. Processes communicate through RPC. Programmers can define atomic objects, which are manipulated in atomic transactions (consisting of multiple RPC calls, possibly involving many different guardians). If a transaction commits, the new state of the atomic objects changed by the transaction is saved on stable storage. After a guardian crashes, it is recovered (possibly on a different processor) by first restoring its atomic objects and then executing a user-defined recovery procedure.

Our parallel Argus programs use multiple guardians, typically one per processor. Each guardian does part of the work and all guardians run in parallel. Guardians occasionally checkpoint important status information, by storing this information in atomic objects.

An important advantage of letting the programmer deal with fault tolerance is the increased efficiency. Our Argus programs, for example, only save those bits of state information that are essential for recovering the program after a failure. They do not checkpoint data structures that the programmer knows will not change, nor do they save temporary (e.g., intermediate) results. In addition, it is frequently possible to recover only the guardian that failed, rather than all guardians (and processes) in the program. Finally, the programmer can control *when* checkpoints are made. For example, if a process has just computed important information that will be needed by other processes, it can write this information to stable storage immediately.

In Orca, programmers do not have these options. On the other hand, programming in Orca is much simpler, because fault tolerance is handled transparently by the system. For the parallel Argus programs, the extra programming effort varied significantly between applications. Some applications were very easy to handle. For example, a program using replicated worker style parallelism [Carriero et al. 1986] merely needs to write the jobs (tasks) of the workers to stable storage. If a worker crashes, the job it was working on is simply given to someone else, similar to the scheme described in [Bakken and Schlichting 1991]. Other applications, however, require much more effort. For parallel sorting, for example, a lot of coordination among the parallel processes is needed to obtain fault tolerance [Bal 1992].

Of course, there are many other language constructs that could be used for fault-tolerant parallel programming. Examples are: exception handlers, fault-tolerant Tuple Space [Xu 1988] and atomic broadcasts. The Amoeba broadcast protocol, for example, can tolerate processor crashes, so it can be used for building fault-tolerant applications [Kaashoek and Tanenbaum 1990]. Little experience in using these mechanisms for parallel programs is reported in the literature, however.

6.2. Other mechanisms providing transparent fault tolerance

Several other systems provide fault tolerance in a transparent way [Strom and Yemini 1985; Sistla and Welch 1989; Johnson 1989; Koo and Toueg 1987]. Most of these schemes are based on message logging and playback, usually in combination with periodic checkpoints. They are much more general than the method we described, in that they can also handle interactive distributed programs and sometimes can even give real-time guarantees about the system.

We compare our approach in more detail with one of the message logging approaches. We have chosen to compare it with Johnson's work [Johnson 1989], because it is implemented on the V system [Cheriton 1988], a system comparable to Amoeba, and he gives performance figures of his implementation. Johnson's approach is much more general than our approach (it can deal with interactions with the outside world) but is also much more complicated and harder to implement. It requires, for example, extensive modifications to the V kernel. Furthermore, it logs every message. This increases the cost for communication substantially (between 22 and 36 percent) and this introduces a fixed overhead for all applications. Using our scheme the overhead depends on the frequency of making checkpoints and is independent of the number of messages that an application sends. For parallel programs, this property is important, since such programs usually communicate a lot.

An interesting system related to ours is that of Kai Li [Li et al. 1990]. This system also makes periodic global checkpoints. Unlike ours, however, it does not delay the processes until the checkpoint is finished. Rather, it makes clever use of the Memory Management Unit. The idea is to make all pages that have to be dumped *read-only*. After this has been done, the application program is resumed and a *copier* process is started in parallel, which writes the pages to disk. Since all pages are read-only to the application, there is no danger of losing consistency. If the application wants to modify a page, it gets a page-fault. The page-fault handler asks the copier process to checkpoint this page first. After this has been done, the page is made writable and the application is resumed. In this way, much of the checkpointing can overlap with the application.

In principle, we could have used this idea for making a checkpoint of a single process, but it would require extensive modifications to the sources of the memory management code and the way checkpoints are made in Amoeba. As we wanted to keep our implementation as simple as possible, we were not prepared to implement this optimization.

Another related approach is that of [Wu and Fuchs 1990]. In this paper, the authors describe a method to make a page based shared virtual memory fault-tolerant. Like our method is their method transparent to the programmer and is intended for long running parallel computation. In their method, the owner process of a modified page takes a checkpoint before sending the page to the process that requests it. Therefore, unlike our method, the frequency of checkpointing is determined by the patterns of data sharing. Frequent checkpointing occures if two process alternately write the same page.

7. CONCLUSION

We have described a very simple method for making parallel Orca programs fault-tolerant. The method is fully transparent, and works for parallel programs that take input, compute, and then yield a result. The method is not intended for interactive programs, nor for real-time applications.

Our method makes use of the fact that processes in the Orca implementation communicate through indivisible reliable broadcasting. In a system based on point-to-point message passing, it would be much harder to make a consistent checkpoint of the global state of the program. One approach might be to simulate indivisible broadcasting, for example by using the algorithm described in [Bal and Tanenbaum 1991]. This algorithm includes timestamp vectors in each message being sent, which are used in determining a consistent ordering. Another method might be to freeze the whole system before a checkpoint is made, but this introduces a performance penalty.

The paper also describes an actual implementation of our system, on top of the Amoeba distributed operating system. We have identified a number of problems with some Amoeba services, in particular the process server. We managed to get around these problems, but the implementation would have been much simpler if certain restrictions in Amoeba were removed. Finally, we have given initial performance results of our system, using a simple parallel application.

ACKNOWLEDGEMENTS

The authors would like to thank Dave Bakken, Leendert van Doorn, Fred Douglis, Elmootazbellah Nabil Elnozahy, Dick Grune, Cees Verstoep, and Cees Visser for their careful reading of the paper.

REFERENCES

Bakken, D. E. and Schlichting, R. D., "Tolerating Failures in the Bag-of-Tasks Programming Paradigm," *Proc. of the 21st International Symposium on Fault-Tolerant Computing*, pp. 248-255, Montreal, Canada, June 1991.

Bal, H. E., "Programming Distributed Systems," Silicon Press, Summit, NJ, 1990.

Bal, H. E., "Fault-tolerant Parallel Programming in Argus," *Concurrency Practice & Experience*, 1992. (accepted for publication)

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "A Distributed Implementation of the Shared Data-Object Model," *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL, Oct. 1989.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Experience with Distributed Programming in Orca," *IEEE CS Int. Conference on Computer Languages*, pp. 79-89, New Orleans, LA, Mar. 1990.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Orca: A language for Parallel Programming of Distributed Systems," *IEEE Transaction on Software Engineering*, 1992. (accepted for publication)

Bal, H. E. and Tanenbaum, A. S., "Distributed Programming with Shared Data," *Comp. Lang.*, Vol. 16, No. 2, pp. 129-146, 1991.

Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Call," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Carriero, N., Gelernter, D., and Leichter, J., "Distributed Data Structures in Linda," *Proc. 13th ACM Symp. Princ. Progr. Lang.*, pp. 236-242, St. Petersburg, FL, Jan. 1986.

Cheriton, D., "The V Distributed System," *Commun. ACM*, Vol. 31, No. 3, pp. 314-333, Mar. 1988.

Douglis, F., Kaashoek, M. F., Tanenbaum, A. S., and Ousterhout, J. K., "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, 1992. (accepted for publication)

Johnson, D. B., "Distributed System Fault Tolerance Using Message Logging and Checkpointing," TR89-101 (Ph.D. thesis), Rice University, Dec. 1989.

Kaashoek, M. F. and Tanenbaum, A. S., "Fault Tolerance Using Group Communication," *Proc. of 4th ACM SIGOPS European Workshop*, Bologna, Italy, Sep. 1990. (Also published in SIGOPS OSR, Vol. 25, No. 2)

Kaashoek, M. F. and Tanenbaum, A. S., "Group Communication in the Amoeba Distributed Operating System," *Proc. The 11th Internatinal Conference on Distributed Computer Systems*, pp. 222-230, IEEE Computer Society, Arlington, TA, May 1991.

Kaashoek, M. F., Van Renesse, R., Van Staveren, H., and Tanenbaum, A. S., "FLIP: an Internetwork Protocol for Supporting Distributed Systems," IR-251, Vrije Universiteit, Dept. of Math. and Comp. Sci., Amsterdam, June 1991. (submitted for publication)

Koo, R. and Toueg, S., "Checkpointing and Roll-back Recovery for Distributed Systems," *Trans. Soft. Eng.*, Vol. SE-13, No. 1, pp. 23-31, Jan. 1987.

Li, K. and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Syst.*, Vol. 7, No. 4, pp. 321-359, Nov. 1989.

Li, K., Naughton, J. F., and Plank, J. S., "Real-Time, Concurrent Checkpoint for Parallel Programs," *Proc. 2nd Symposium on Principles and Practice of Parallel Programming*, pp. 79-88, Seattle, WA, Mar. 1990.

Liskov, B., "Distributed Programming in Argus," *Comm. ACM*, Vol. 31, No. 3, pp. 300-312, Mar. 1988.

Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staveren, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23, No. 5, pp. 44-53, May 1990.

Sistla, A. P. and Welch, J. L., "Efficient Distributed Recovery Using Message Logging," *Proc. 8th ACM Symp. Princ. of Distr. Comp.*, pp. 223-238, Edmonton, Alberta, Aug. 1989.

Strom, R. and Yemini, S., "Optimistic Recovery in Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 3, No. 3, pp. 204-226, Aug. 1985.

Tanenbaum, A. S., Kaashoek, M. F., and Bal, H. E., "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, 1992. (accepted for publication)

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S., Jansen, A., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," *Commun. ACM*, Vol. 33, No. 12, pp. 46-63, Dec. 1990.

Van Renesse, R., Tanenbaum, A. S., and Wilschut, A., "The Design of a High-Performance File Server," *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, pp. 22-27, Newport Beach, CA, June 1989.

Wu, K-L. and Fuchs, W. K., "Recoverable Distributed Shared Virtual Memory," *IEEE Trans. on Comp.*, Vol. 39, No. 4, pp. 460-469, Apr. 1990.

Xu, A. S., "A Fault-tolerant Network Kernel for Linda," TR-424, M.I.T., Cambridge, Mass., Aug. 1988.

APPENDIX

In this section we will deduce the formula to compute the overhead of checkpointing and the optimal computing interval given the time to make a checkpoint, the time to recover from a crash, and the MTTF of the system.

For the derivation we introduce the following variables:

T_{tot} is the time needed to run the application without checkpointing;
T_{comp} is the time between two checkpoints;
T_{cp} is the mean time to make a global checkpoint;
T_{rb} is the mean time to roll back;
T_{MTTF} is the mean time to failure.

T_{comp}	T_{cp}	T_{comp}	T_{cp}	T_{comp}	T_{cp}	T_{comp}	T_{cp}

Fig. 3. An Orca execution without failures.

An execution of an Orca application without any failures consists of a sequence of alternating computing intervals of length T_{comp} and checkpoint intervals of length T_{cp} (see Fig. 3). If a failure happens during a computing interval or a checkpoint interval, an extra interval with duration equal to the recover time plus the time wasted before the crash is inserted (see Fig. 4). The mean time wasted can be estimated by $(T_{comp} + T_{cp}) / 2$.

T_{comp}	T_{cp}	$(T_{comp}+T_{cp})/2$	T_{rb}	T_{comp}	T_{cp}	T_{comp}	T_{cp}

Fig. 4. An Orca execution with a failure.

Let T_{tot} be the total computation time needed to finish the application. Then, the number of computing intervals is equal to T_{tot} / T_{comp} and the number of failures during the total run time is approximately T_{tot} / T_{MTTF} (assuming $T_{cp} << T_{tot}$ and $T_{rb} << T_{tot}$). Thus the average run time is:

$$\frac{T_{tot}}{T_{comp}} (T_{comp} + T_{cp}) + \frac{T_{tot}}{T_{MTTF}} (\frac{T_{comp} + T_{cp}}{2} + T_{rb}).$$

The overhead for checkpointing is the average run time divided by T_{tot} minus 1:

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{\frac{T_{comp} + T_{cp}}{2} + T_{rb}}{T_{MTTF}}.$$

Minimizing the overhead function gives:

$$optimal\ computing\ interval = \sqrt{2T_{cp}T_{MTTF}}$$

CLASS LIBRARIES AS AN ALTERNATIVE TO LANGUAGE EXTENSIONS FOR DISTRIBUTED PROGRAMMING

Thomas G. Dennehy
AT&T Bell Laboratories[1]
Whippany, NJ 07981

ABSTRACT

Designing distributed processing systems requires specifying not only the computations to be performed by independent activities but also how these activities coordinate and communicate. While a number of languages with "parallel extensions" have been invented to provide both a computation and coordination model, we show in this paper how class libraries in an object-oriented language like C++ offer a better alternative for distributed programming. Parallel programs are very naturally described by object-oriented models, and this formulation can be used to effect a clean separation between application code (the computation model) and system code (the coordination model), creating software that can be readily ported across different hardware architectures. The software architecture of an adaptive Image Understanding System (IUS) and the design process leading to eventual deployment on a dynamic dataflow machine operating with a multiple-instruction multiple-data (MIMD) parallel processor front-end are described. At the core of the design is a small set of classes defining the interface between the application domain and the system domain, enabling the IUS design to be instantiated several different ways in various environments by replacing system components without affecting application code. In addition to portability, the positive impact of this design approach on development time, code size, localization of hardware influences, and defect removal are also discussed.

1. Introduction

The integrated operation of the AN/UYS-2 Enhanced Modular Signal Processor (EMSP), a dynamic dataflow machine, and the AT&T DSP3, a multiple-instruction multiple-data (MIMD) parallel processor, has been demonstrated by building the prototype of an adaptive Image Understanding System (IUS). The IUS software architecture provided a rich framework for studying critical systems/software design issues for applications to be deployed on networks of heterogeneous processors. In particular, the choice of language proved key for creating a portable IUS architecture and for using the porting process as an effective defect removal technique during development.

To write distributed programs, one must be conversant in two distinct vocabularies. First there is the vocabulary of the problem domain, or the *computation model*; software in a distributed program embodying the computation model will hereafter be called *application code*. Second, there is the vocabulary of the system domain, or the *coordination model*; software embodying the coordi-

1. Author's Present Address: Environmental Research Institute of Michigan, P. O. Box 134001, Ann Arbor, MI 48113-4001. Electronic mail to `dennehy@erim.org`.

nation model will hereafter be called *system code*. Program decomposition in the problem domain involves choosing the right granularity and topology of software components, creating a virtual distributed processing network.[2] Program composition in the system domain is a two-part task: 1) allocating hardware resources, perhaps widely separated in space or time, to satisfy the computation, memory, and communication requirements of the software components; and 2) expressing how to create and control multiple execution threads.[3] While a number of languages with ''parallel extensions'' have been invented to provide both a computation and coordination model, we show in this paper how class libraries in an object-oriented language like C++ offer a better alternative for distributed programming.

2. Class Libraries v. Language Extensions

Our goal is to provide a design vocabulary for system code such that the application interface to system components is independent of the physical process distribution and communication channels used, creating distributed programs that may be conveniently ported across different operating environments or hardware architectures. To achieve this goal, our design method must:

1. **Provide for convenient reuse of existing (application) code.** It is becoming common to see application building interfaces for specific domains (rendering, signal or image processing, etc.) through which a user can create a processing network by connecting prepackaged components. A good example of a visual language for connecting somewhat arbitrary application modules is ConMan.[2]

2. **Provide for different models of system code.** Too often, however, such application builders provide a limited and non-extensible number of process and communication models, typically UNIX® system processes communicating through pipes or sockets. Since we are concerned with networks of heterogeneous components, a richer system vocabulary is required. Not only must an extensible application interface to support a range of synchronization and communication styles be provided, it must also be convenient to replace system components to improve performance without disturbing application code.

3. **Provide for different implementations of system code on different architectures.** In addition, it should be feasible to rehost software from one operating environment to another environment that provides similar services without replacing system components merely through recompilation or relinking.

A distributed programming language which satisfies these requirements must accommodate differ-

2. In fact, Booch[1] has stated that the start of the object-oriented design process is to "discover the classes and objects that form the vocabulary of our problem domain. The tangible things in the problem domain, the roles they play, and the events that occur form the candidate classes and objects of the design." (p. 190-191).

3. The classes and objects that form the vocabulary of the system domain must explain how events in the problem domain are brought about and how the performance requirements of a given installation of a design are met.

® UNIX is a registered trademark of Unix System Laboratories, Inc.

ent styles of parallelism both in the algorithms and the hardware network.

A number of general-purpose languages for distributed programming have been designed to provide both a computation and coordination model. *Linda*[3] is an independent model of process creation and coordination embedded in a base language. Programming environments have been created by embedding Linda operations in common languages like C and Fortran. *Concurrent C* is a superset of C for UNIX systems that provides parallel programming facilities to control multiple user processes and transactions among them.[4] *C** is a data parallel superset of C originally designed by Thinking Machines (TM) for the Connection Machine Processor Array. Dialects have been produced by TM[5] and jointly by the University of New Hampshire and Oregon State University for multiprocessors from nCUBE and Intel,[6] and also Sequent.[7] In addition, there are numerous proprietary dialects of common languages designed for specific parallel architectures.

Parallel extensions to sequential programming languages introduce a coordination model through new syntax, keywords, and/or system calls, fostering a programming style that tightly couples application code and system code rather than defining a clean interface between the two. In fact, the style has been described as equivalent to programming in C with escapes to assembly language,[8] with predictable results: the code may perform well on a particular architecture, but is almost certainly non-portable. Furthermore, these languages usually embody one parallel programming model, and can only be used effectively if the underlying model is well-matched to the inherent parallelism of the application. Even if a ''general'' model is provided— Linda's generative communication model can be implemented using a variety of physical channels and protocols[9] — the vocabulary provided is not rich enough to access a variety of styles in any single application.

Object-oriented techniques provide a bridge from the sequential world into the parallel world without new languages. Parallel programs are very naturally described by object-oriented models, and this formulation provides excellent hints to the system for partitioning software components across hardware networks. Class vocabularies can be created to invent design languages appropriate for both application code and system code and to effect a well-defined interface between the two. Class libraries can also be used to create ''wrappers'' for system code written in special-purpose languages, particularly C-based parallel languages. Employing object-oriented techniques for distributed programming creates software that is readily portable, application components that can be used different ways in different contexts, and convenient encapsulations for different implementations of a variety of synchronization and communication styles.

In the next section, we present the architecture of an adaptive Image Understanding System (IUS) and describe a design process using the C++ language leading to the deployment of an IUS prototype on a heterogeneous hardware network. The design process ensures software portability; the IUS was, in fact, ported to three different environments during development. Positive effects on development time, code size, and the isolation of hardware artifacts are also discussed.

3. Case Study: An Image Understanding System

The software architecture of an adaptive Image Understanding System (IUS) can be decomposed into three subsystems: System Control, Pixel Processing, and Segment Processing (figure 1)). System Control initializes the IUS and handles user interaction. Pixel Processing handles segmentation of the input image, passing segments to Segment Processing, which assembles an image description and updates certain parameters in the Pixel Processing pipeline to adapt processing to ambient conditions.

Pixel Processing is an *N*-stage pipeline of segmentation operations — noise removal, filtering, thresholding, and so on— characterized by a constant stream of input, low-latency processing, and a fixed communication pattern between stages (figure 2). In addition to pipelining, Pixel Processing can be further parallelized by allocating the pixel space across several instances of each stage. A control task monitors pipeline operation through test points labeled 1-N. This style of parallel computation is well-suited to a systolic array processor.

In addition to building the image description, Segment Processing performs its parameter update calculations independently in a set of zones that partition the image space (figure 3). The input volume is much lower that of Pixel Processing, and the computation load per zone is data-dependent. This style of parallel computation is well-suited to a dataflow machine.

Practical high-resolution applications of this IUS architecture can have real-time requirements that stress the capabilities of available hardware. For example, an Infrared Search and Track system based on this approach[10] has an input rate of roughly 10M pixels/sec and processing requirements of over 500 MFLOP/sec. To meet these requirements, a processor network was chosen with the following components:

• A Sun-3 Workstation running SunOS 4.0.3 for system control and operator displays.[4]

An AT&T DSP3 Parallel Processor for Pixel Processing. The DSP3 is an MIMD processor composed of two major subsystems (figure 4):

— The DSP3 Real-Time Host (DSP3-RTH), a 680x0-based single-board computer running the VRTX multi-task kernel from Ready Systems; and

— The DSP3 Data Subsystem (DSP3-DS), an array of 16-128 Processing Elements (PE), each PE composed of a 25 MFLOP DSP32C signal processor, 256 Kbytes of local memory, and a configurable communication device. A minimal software kernel operates in each PE to handle remote procedure execution from the DSP3-RTH.

• An AN/UYS-2 Enhanced Modular Signal Processor (EMSP) for Segment Processing. The EMSP dataflow architecture is a loosely-connected network of processors, with 8 functional ele-

4. Sun and SunOs are registered trademarks of Sun Microsystems, Inc.
VRTX is a registered trademark of Ready Systems.

ment types serving as the modular building blocks of the architecture (figure 5). Each Arithmetic Processor (AP) FE can deliver 120 MFLOP. The open architecture of EMSP allows integration of application-specific functional elements; the DSP3 is operating as a so-called "matrix processor" for EMSP in the IUS.

In designing the IUS, the approach taken was to specify software components using C++ class definitions, creating a complete and compilable interface specification for each component independent of its eventual implementation language or target hardware. These class definitions were then refined to create a behavioral simulator of the IUS (using the C++ task library to represent the multiple execution threads) used to verify system operation and resolve initialization, resource distribution, and message handling issues. Once the optimal component structure of the IRSS-SP had been identified via simulation, components were "broken out" and assigned to target hardware, using the simulation code as a baseline implementation wherever feasible.[5]

Classes representing the inherent parallelism of IUS Pixel Processing (the application code) were derived from two bases:

`TestPointControl`: The point of contact between the user and the parallel execution threads that implement stages in the Pixel Processing pipeline is the `TestPointControl`, whose responsibilities include distributing commands to the various tasks associated with it and assembling task reports sent in response.

`Stage`: A `Stage` represents common behavior among the stages in the Pixel Processing pipeline, receiving messages and handling Test Point operations.

Encapsulating the parallelism of Pixel Processing in `TestPointControls` and `Stages` produced a software architecture that could be readily adapted to the structure of the input. Classes derived from `TestPointControl` control the number of parallel tasks created for each processing stage and how the image space is divided across these tasks, and control how incoming messages are routed to one or more tasks, so that the message handling responsibilities of the Pixel Processing Control task could be implemented simply by passing any message to the appropriate `TestPointControl` to `Handle()`. Classes derived from `Stage` are customized by the processing algorithm they perform in response to messages identified as incoming data. Thus, any of the segmentation algorithms can be modified or replaced without impact on the architecture, and all `Stages` resident on a particular hardware component can share a large body of code.

Classes representing the parallelism of the hardware network (the IUS system code) were derived from two bases:

`Medium`: The class `Medium` provides a uniform interface to the various physical I/O media used in the processor network, identifying a connection between two processes while isolating the details of making that connection. Virtual functions defined for class `\Medium`

5. Since no C/C++ translation system exists for the EMSP, a different implementation methodology was used.

(`read()`, `write()`, `poll()`, etc., providing service at the roughly the level of UNIX I/O system calls) are the application code interface to physical I/O.

`Activity`: The class `Activity` provides a uniform interface to various methods for establishing independent processing activities while isolating the details of initiating execution.

Inter-component interfaces based on classes `Activity` and `Medium` create a vocabulary for assigning communicating software components to hardware elements. The communication interface between any `Activity/Medium` configuration and application code is based on a small set of virtual functions, so that portable application-level message protocols can be devised independent of the physical process distribution and communication channels actually used.

For example, the migration of the IUS components across the processor network during system initialization is defined in terms of one process instantiating another process by means of the class `RemoteControl`, which pairs an independent `Activity` with a `Medium` for communicating with it. In this model, one step in the instantiation of the IUS in a particular environment might be to create a prefiltering step as a `RemoteStage` and to establish communication through a `Bus-Interface` to a DSP3 internal bus. Porting this configuration to a different environment requires changing the vocabulary used in the constructor, perhaps instantiating a `CoProcess` and communicating through a `UnixSocket`, but not changing the application interface to class `Prefil-ter`.

```
class RemoteControl {
            ...
        class Medium* command;
        class Activity* control;
            ...
};

class Prefilter :
public RemoteControl { ... };
```

```
Prefilter::Prefilter( ...) :
RemoteControl(
            ...
        new RemoteStage(...),
        new NetworkInterface(...),
            ...
) { ... }
```

While the classes `TestPointControl` and `Stage` embodying the inherent parallelism of the IUS may not be broadly applicable, a rich vocabulary of synchronization and communication styles based on `Activity` and `Medium` was created that can serve as the foundation of a design toolkit. The class vocabulary of communication can be partitioned into two groups (figure 6). The devices in Group A are either application-specific or represent resources common to many platforms; systems incorporating this vocabulary can be ported merely by recompilation (if necessary) and re-linking with appropriate implementations. The devices in Group B are platform-specific; systems incorporating this vocabulary can be ported by selecting an appropriate device for the chosen platform, requiring modification to system code that handles initialization but no changes to application code.

The portability of designs based on this systems vocabulary was demonstrated during prototype development, as the IUS was instantiated in three different environments by replacing system components without affecting application code.

1. The first generation of the IUS was its behavioral simulator running as a single "heavy-weight" process on the Sun-3. All execution threads were lightweight processes and all communication media were based on inter-thread FIFOs.

2. The second generation of the IUS extracted Pixel Processing from the simulator and hosted it on the DSP3. The control task was ported to the DSP3-RTH while the processing pipeline was ported to the processor mesh (DSP3-DS), with groups of PEs allocated to each pipeline stage. Our synchronization vocabulary was extended to include VRTX task creation and remote procedure call between the DSP3-RTH and DSP3-DS. The communication vocabulary was extended to include Ethernet sockets, VRTX FIFOs, and the internal communication paths of the DSP3.

3. The third generation of the IUS used the same hardware suite, but reorganized the DSP3-DS so that each PE was running a copy of the entire Pixel Processing pipeline in order to gain communication efficiency. In this case, the pipeline stages were not independent tasks at all, merely software objects, and a communication "device" was designed that passed a message by calling the message handler of the receiver. Since this was the IUS actually delivered, a scaffolding of graphic displays was erected around this version, requiring us to extend our system vocabulary to include the fork/exec of heavyweight processes and communication through `UNIX` sockets.[6]

By making it easy to port the IUS to different environments, we were able to take advantage of porting as an effective defect removal technique during development, improving the quality of the system.[11]

In addition to creating a truly portable distributed system, our design approach using the C++ language paid other substantial benefits:

1. **Short Development Time**: The IUS software, consisting of 12 KSLOC of C++, was produced over a period of six months. Most of that time was devoted to class design and creating the simulation; the transition from baseline simulator to third generation system on target hardware, including detailed implementation of the segmentation operations, took less than 10 weeks.

2. **Component Re-use**: 92 of the 139 class declarations in the IUS were used on two or more

6. As a further demonstration of software re-use by replacing system components, one version of the display programs was created that ran interactively with the IUS and another version was created to run in "playback" mode, reading data from files. The versions differed only in the communication media used.

hardware platforms (table 1). Of the 37 class declarations used on one hardware platform, 23 are designed for that platform only, and, of those 23, 22 represent either process synchronization (derived from `Activity`) or communication (encapsulating hardware resources or derived from class `Medium`). For example, just 1 of 11 Core classes used only on the Sun-3 is designed specifically for a `UNIX` system platform (the `CoProcess` class); the other 10 classes embody the functionality of the System Control Process, which, although hosted on a Sun in this version of the IUS, could be deployed on other hardware

TABLE 1. IUS CLASS USAGE PER HARDWARE PLATFORM

		Used on One Platform			*Used on Two*
	Classes	*Sun-3*	*DSP3-RTH*	*DSP3-DS*	*or More Platforms*
`/include`	82	11 (1)	8 (1)	3 (0)	60
`/include/sys`	22	0	1 (1)	0	21
`/include/media`	35	9 (7)	8 (7)	7 (6)	11
	139	20 (8)	17 (9)	10 (6)	92

3. **Reduced Code Size**: Code size reduction of over 40% was achieved by re-use of system components across hardware platforms (table 2).

TABLE 2. IUS SOURCE CODE STATISTICS.

	Source Lines of C++			
	Sun-3	*DSP3-RTH*	*DSP3-DS*	*Total (Unique)*
`/src`[a]	1926	2584	4110	8620 (6673)
`/lib`	834	1832	460	3126 (2820)
`/include`	874	1335	1230	3439 (1673)
`/include/sys`	306	373	320	999 (373)
`/include/media`	343	460	265	1068 (575)
	4283	6584	6385	17252 (12114)

a. `/src`	IUS Core functionality.
`/lib`	Implementation of platform-specific system resources.
`/include`	Declarations for IUS Core functionality.
`/include/sys`	Declarations for IUS system parameters.
`/include/media`	Declarations for communication media.

4. **Minimization of Hardware Impact**: Excluding implementations of process models and communication media, less than 8% of the source code in the IUS is platform-specific. 70% of the source application code is devoted to core functionality.

4. Conclusions

Other approaches based on object-oriented design techniques have emphasized reliability rather that portability. Arjuna[12] and Avalon[13] are two distributed programming systems that use C++ to provide mechanisms for invoking atomic actions on objects. Both are monolithic systems, limiting portability and the capacity for program performance optimization via choice of process models or communication media. Aeolus[14] is a systems programming language providing abstractions of the objects, actions, and processes of the Clouds[15] distributed operating system. Using C++ to provide these abstractions (rather than a new language like Aeolus) would not only enable applications incorporating the reliability features of Clouds to be run in other environments but also demonstrate — without modifying application code — how the performance of those programs could be improved by implementing reliability at the kernel level.

Using class libraries in C++ instead of a language with parallel extensions enabled us to achieve portability in distributed systems design by creating appropriate vocabularies for both computation and coordination models, effecting a clean separation between application code and system code. While the classes embodying the inherent parallelism of the IUS may not be broadly applicable, a rich vocabulary of communication and synchronization styles was created that can serve as the foundation of a design toolkit. By making it easy to port software systems and employing a design method based on porting code to different environments, defects exposed by hardware, operating system, compiler, and environment differences are found during initial development, making it more likely that software components can be re-used.

REFERENCES

[1] Booch, Grady, *Object-Oriented Design with Applications*, The Benjamin Cummins Publishing Company, Redwood City, CA, 1991.

[2] Haberli, P., "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics*, **XXII-4**, August, 1988, pp. 103-111.

[3] Ahuja, S., N. Carriero, and D. Gelertner, "Linda and Friends," *IEEE Computer*, **XIX-8**, August, 1986, pp. 26-34.

[4] Gehani, N. and W. Roome, *The Concurrent C Programming Language*, Silicon Press, Summit, NJ, 1989.

[5] Rose, J. and G. Steele, *C*: An Extended C Language for Data Parallel Programming*, Technical Report PL 87-5, Thinking Machines Corporation, Cambridge, MA, 1987.

[6] Hatcher, P. J., et. al., "A Production Quality C* Compiler for Hypercube Multicomputers," *SIGPLAN NOTICES*, **XXVI-7**, July, 1991, pp. 73-82.

[7] Quinn, M., P. Hatcher, and B. Seevers, "Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor," in Nicaula, A., D. Gelertner, T. Gross, and D. Padus (eds.) *Advances in Languages and Compilers for Parallel Processing*, MIT Press, 1991, pp. 385-401.

[8] Weems, C. C. and J. H. Burrill, "The Image Understanding Architecture and its Programming Environment," in Kurma, V. K. P. (ed.) *Parallel Architectures and Algorithms for Image Understanding*, Academic Press, Boston, MA, 1991, pp. 525-562.

[9] Carriero, N. and D. Gelertner, "Linda in Context," *Communications of the ACM*, **XXXII-4**, April, 1989, pp. 444-458.

[10] Lee, R. S. Law, H. Takagi, and T. Dennehy, "A New Signal Processor Architecture for Infrared Search and Track (IRST) Systems," *SPIE Conference on Signal and Data Processing of Small Targets 1992*, to appear.

[11] Fox, C., *Porting for Software Quality*, Technical Memorandum, AT&T Bell Laboratories, August, 1991.

[12] Shrivastava, S. K., G. N. Dixon, and G. D. Parrington, "An Overview of the Arjuna Distributed Programming System," *IEEE Software*, **VIII-1**, January, 1991, pp. 66-73.

[13] Herlihy, M. P. and J. M. Wing, "Avalon: Language Support for Reliable Distributed Systems," *IEEE Computer Society Seventeenth International Symposium on Fault-Tolerant Computing*, July, 1987, pp. 89-94.

[14] Wilkes, C. T. and R. L. LeBlanc, "Rationale for the Design of *Aeolus*: A Systems Programming Language for an Action/Object System," *IEEE Computer Society International Conference on Computer Languages*, October, 1986, pp. 107-122

[15] Allchin, J. E., *An Architecture for Reliable Decentralized Systems*, Technical Report GIT-ICS-83/23, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983.

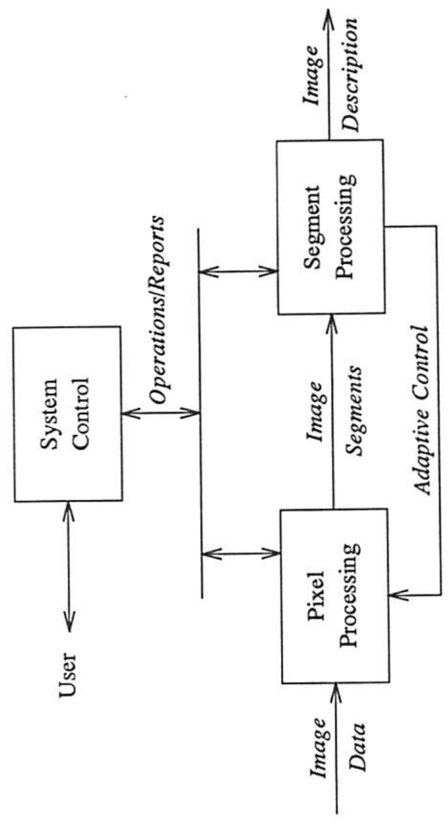


FIGURE 1. AN ADAPTIVE IMAGE UNDERSTANDING ARCHITECTURE

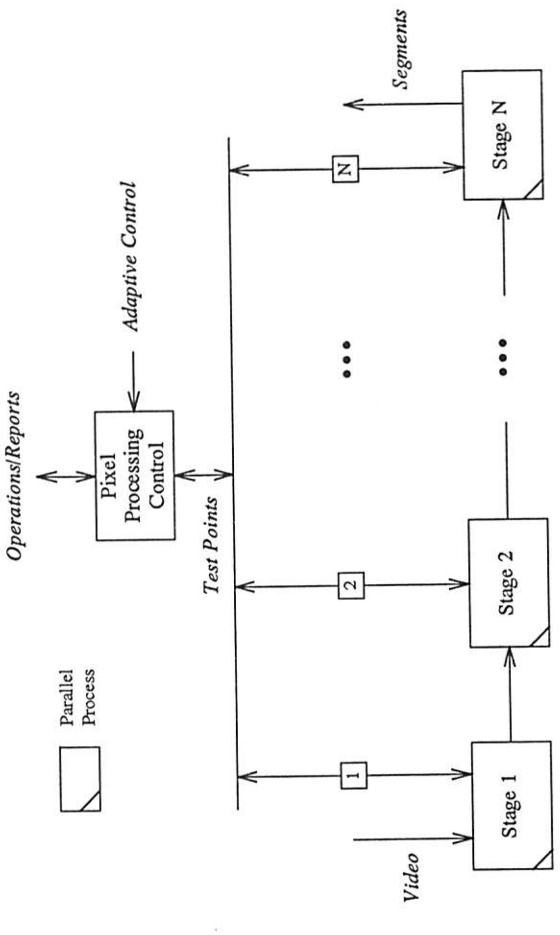


FIGURE 2. PIXEL PROCESSING PIPELINE

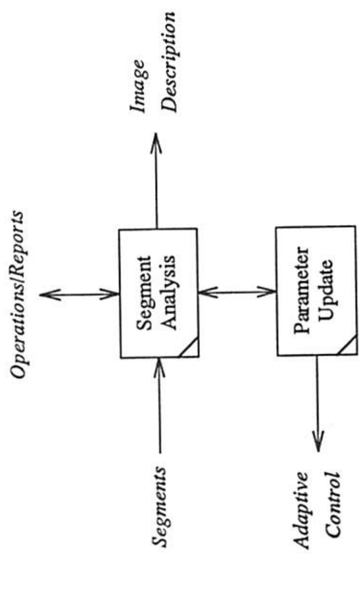
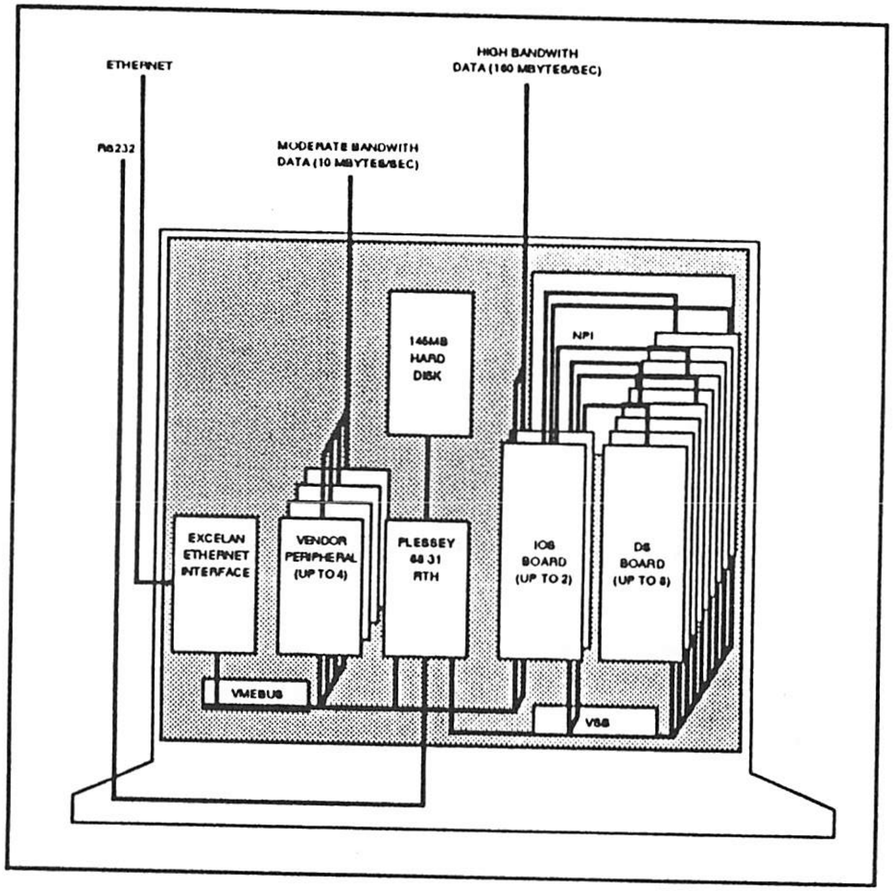
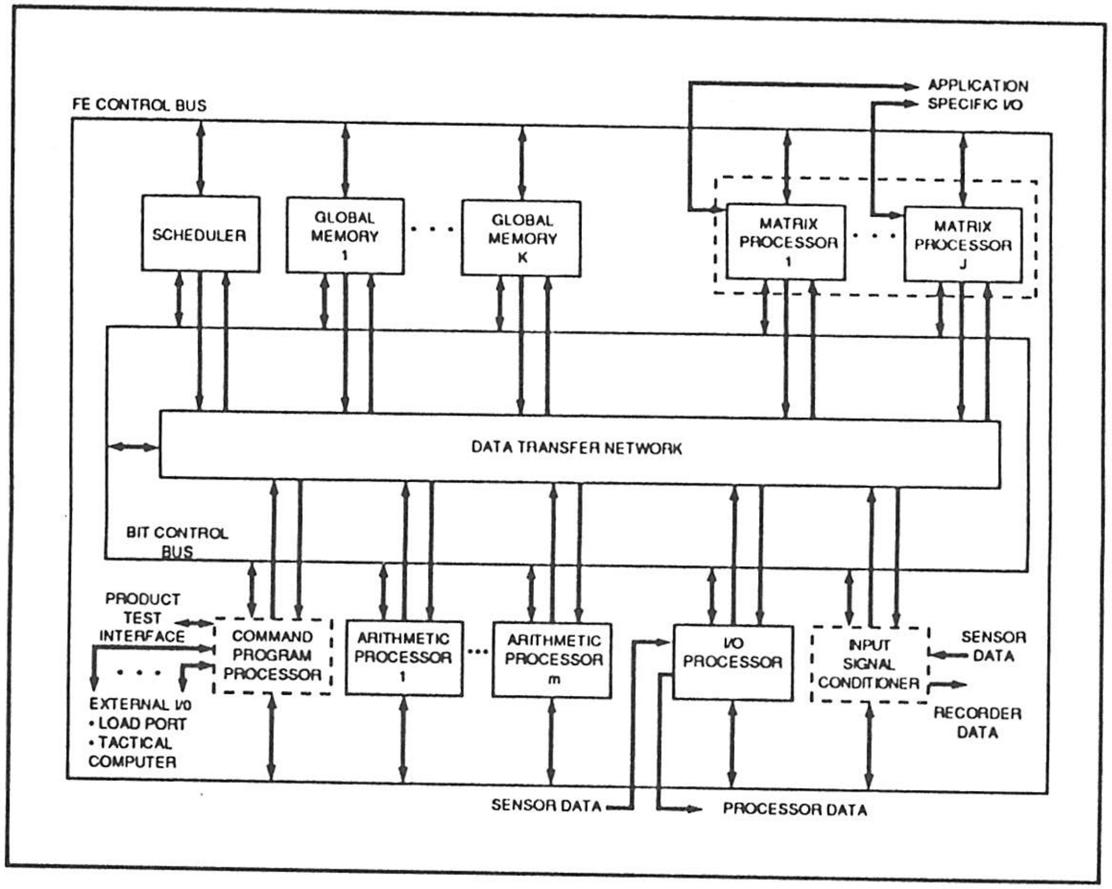


FIGURE 3. SEGMENT PROCESSING


ETHERNET
RS232
MODERATE BANDWITH DATA (10 MBYTES/SEC)
HIGH BANDWITH DATA (160 MBYTES/SEC)
145MB HARD DISK
NPI
EXCELAN ETHERNET INTERFACE
VENDOR PERIPHERAL (UP TO 4)
PLESSEY 68 31 RTH
IOS BOARD (UP TO 2)
DS BOARD (UP TO 8)
VMEBUS
VSB

- MIMD PARALLEL PROCESSOR
- ARCHITECTURAL FEATURES
  - RTH: LOCAL CONTROL PROCESSOR
  - IOS: SUPPORTS CUSTOM HIGH-SPEED AND STANDARD LOW-SPEED INTERFACES
  - DS: 16-128 25MFLOP PROCESSING ELEMENT (PE)
- FUNCTIONS AS A VERSATILE PERIPHERAL COMPUTER FOR REAL-TIME APPLICATIONS

FIGURE 4. DSP3 ARCHITECTURE.



- MODULAR DATAFLOW PARALLEL PROCESSOR
- ARCHITECTURAL FEATURES
  - LOOSELY-CONNECTED NETWORK OF PROCESSORS; 8 FE TYPES
  - EACH AP CAN DELIVER 120 MFLOP/SEC
  - SUPPORTS STANDARD DOD I/O CHANNELS
  - RELIABLE MILITARIZED DESIGN
- OPEN ARCHITECTURE ALLOWS INTEGRATION OF APPLICATION-SPECIFIC FEs

FIGURE 5. AN/UYS-2 EMSP ARCHITECTURE.

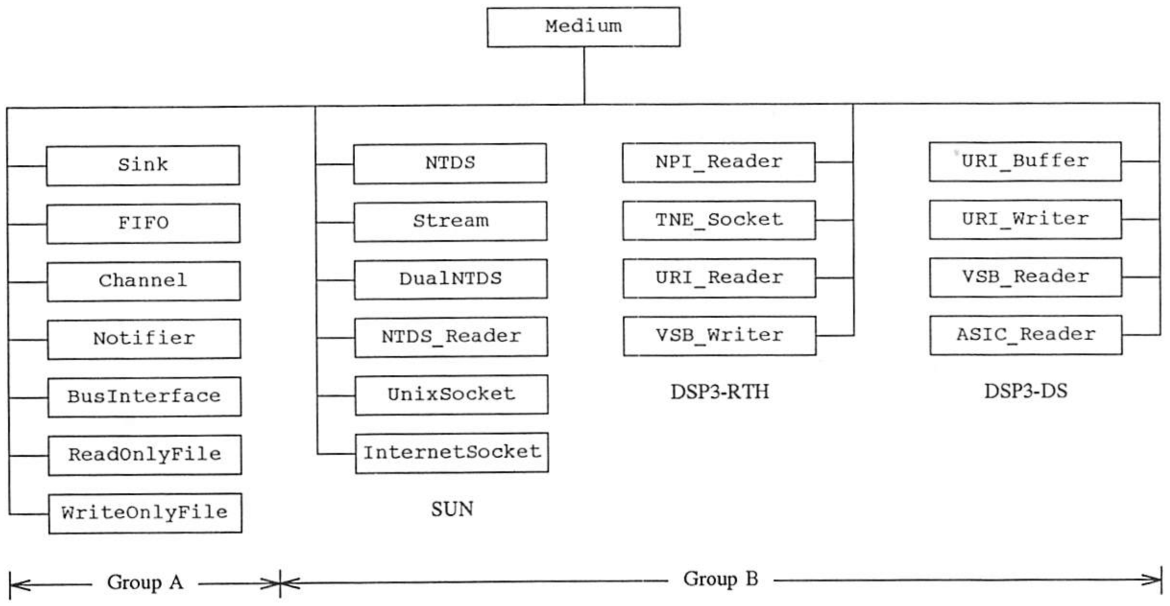


FIGURE 6. CLASS VOCABULARY OF COMMUNICATION MEDIA.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the USENIX Association are:

AT&T Information Systems
Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

ISBN 1-880446-41-3